CSE211 Digital Design

Akdeniz University

Week7-8-9: Gate Level Minimization

Assoc.Prof.Dr. Taner Danışman

tdanisman@akdeniz.edu.tr

Course program

Week 01	2-Oct-23	Introduction
Week 02	9-Oct-23	Digital Systems and Binary Numbers I
Week 03	16-Oct-23	Digital Systems and Binary Numbers II
Week 04	23-Oct-23	Boolean Algebra and Logic Gates I
Week 05	30-Oct-23	Boolean Algebra and Logic Gates II
Week 06	6-Nov-23	Gate Level Minimization
Week 07	13-Nov-23	Karnaugh Maps
Week 08	20-Nov-23	Midterm
Week 09	27-Nov-23	Karnaugh Maps
Week 10	4-Dec-23	Combinational Logic
Week 11	11-Dec-23	Combinational Logic
Week 12	18-Dec-23	Timing, delays and hazards
Week 13	25-Dec-23	Synchronous Sequential Logic
Week 14	1-Jan-24	Synchronous Sequential Logic

Chapter Terms to know

- Karnaugh Map
- Wired Logic
- Don’t care conditions
- HDL

Gate Level Minimization

- *Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.
- Simplification of Boolean functions leads to simpler (and usually faster) digital circuits.
- The Problem:
  - Difficulty in logic design when the problem has more than a few inputs
  - It lacks specific rules to predict the successive steps in the simplification process
  - Simplifying Boolean functions using identities is time-consuming and error-prone.
- Why learning?
  - it is important that a designer understand the **underlying mathematical description** and solution of the problem

The Map Method

- When unique Truth tables represented in algebraically, it can appear in many different, but equivalent, forms.
- Minimization is a difficult process.
- The **Map** method:
  - The **Map** method presented here provides a simple, straightforward procedure for minimizing Boolean functions.
  - Also called pictorial form of a truth table
  - also known as the **Karnaugh map** or **K-map** .

K-map

- A K-map is a diagram made up of **squares**, with each square representing one **minterm** of the function that is to be minimized.
- The map presents a visual diagram of **all possible ways** a function may be expressed in standard form
- By recognizing various patterns, the user **can derive alternative algebraic expressions** for the same function, from which the **simplest can be selected**

K-map

- The simplified expressions produced by the map are always in one of the two standard forms:
  - **sum of products** or **product of sums**
- It will be assumed that the simplest algebraic expression is:
  - an algebraic expression with a **minimum number of terms** and with **the smallest possible number of literals** in each term.
  - This expression produces a circuit diagram with a **minimum number of gates** and the **minimum number of inputs** to each gate
- Sometimes simplest expression is not unique:
  - two or more expressions may satisfy the minimization criteria.
    - In such cases, either solution is satisfactory.

Description of Kmaps and Terminology

- For example, the minterms for a function having the inputs x and y are: $F(x,y) = xy + x\bar{y}$
- Consider the Boolean function,
- Its minterms are:

$\bar{x}\bar{y}, \bar{x}y, x\bar{y},$ and xy

Minterm	X	Y
$\bar{X}\bar{Y}$	0	0
$\bar{X}Y$	0	1
$X\bar{Y}$	1	0
XY	1	1

Description of Kmaps and Terminology

- A Kmap has a cell for each minterm.
- This means that it has a cell for each line for the truth table of a function.
- The truth table for the function *F(x,y) = xy* is shown at the right along with its corresponding Kmap.

F(X,Y) = XY

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X \ Y	0	1
0	0	0
1	0	1

K-Map Rules

- The number of adjacent squares that may be combined must always represent a number that is a **power of two, such as 1, 2, 4, and 8**
- A larger number of adjacent squares are combined, we obtain a product term with fewer literals

1 square = 1 minterm = three literals.

2 adjacent squares = 1 term = two literals.

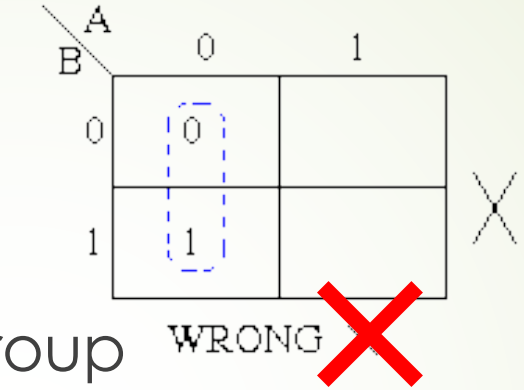
4 adjacent squares = 1 term = one literal.

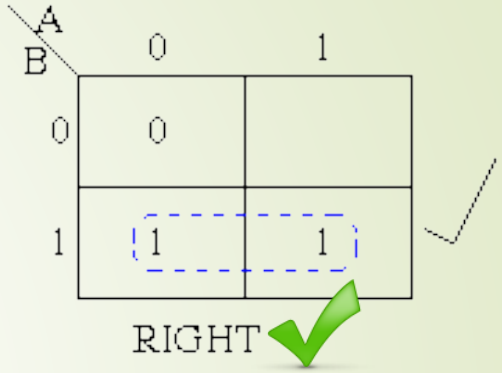
8 adjacent squares encompass the entire map and produce a function that is always equal to 1.

- So it is obviously to know the number of adjacent squares is combined in a power of two such as 1,2,4, and 8.

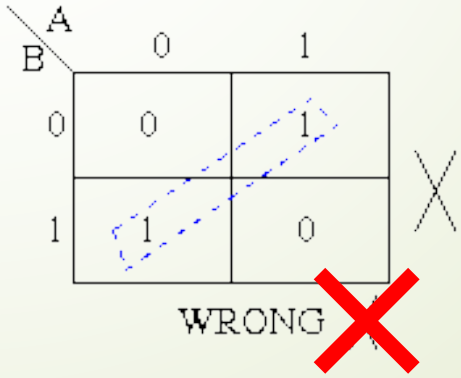
K-Map Rules

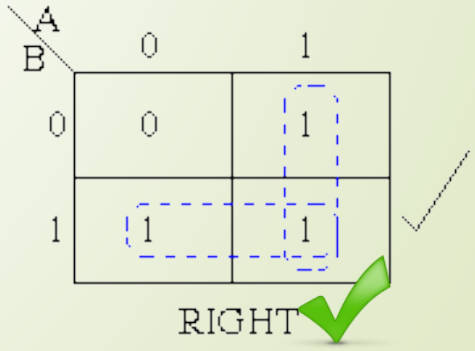
- No zero inside the group





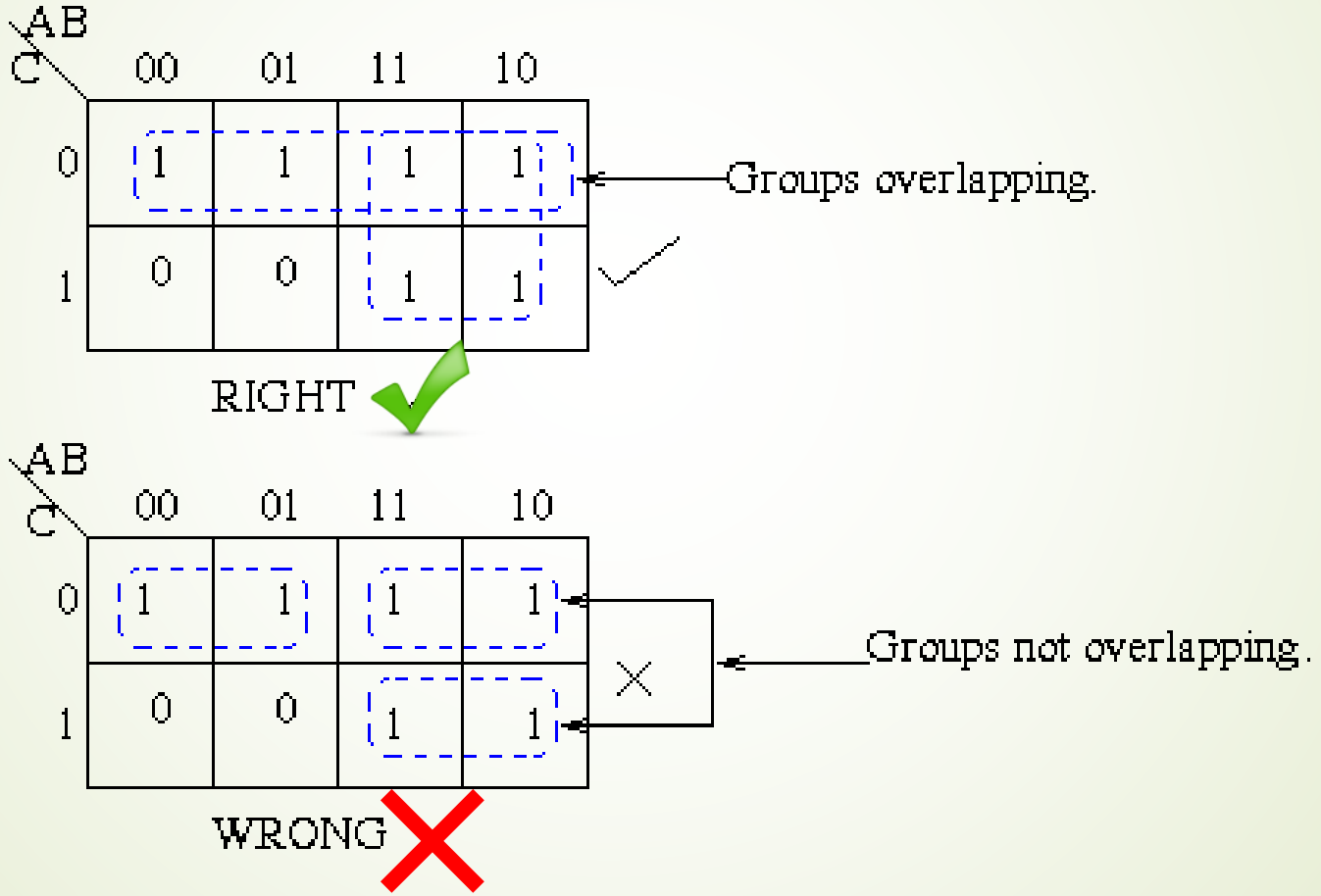
- Groups can be vertical or horizontal but **not diagonal**





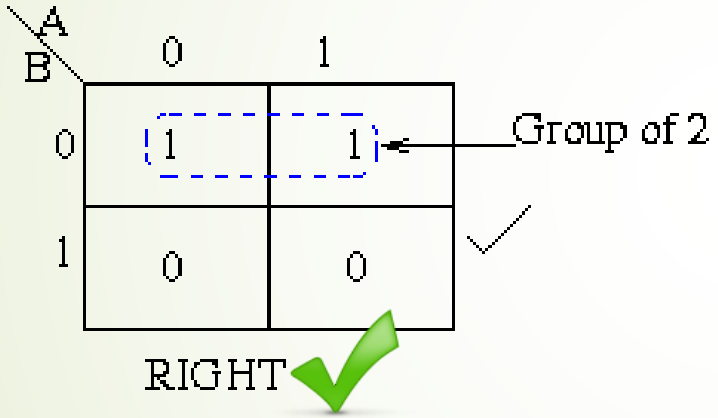
K-Map Rules (Cont.)

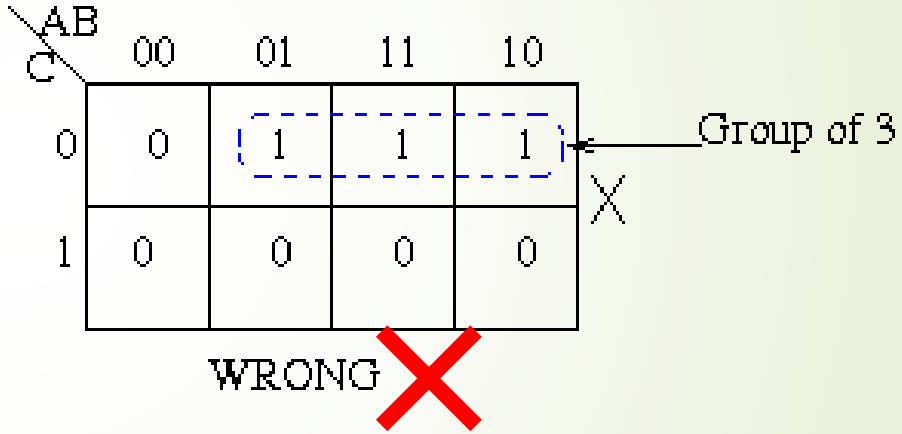
- Groups must be overlapped

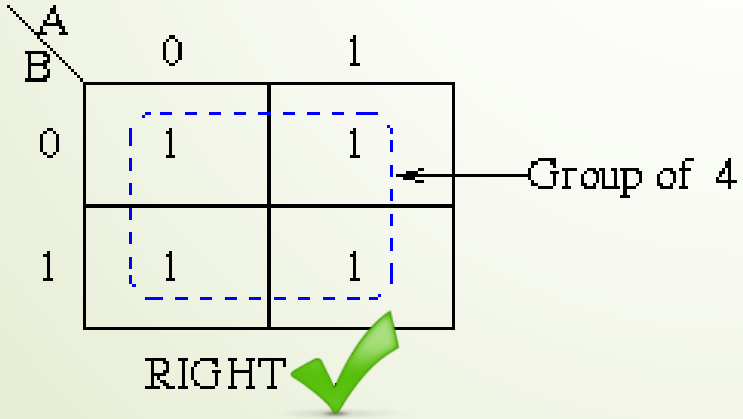


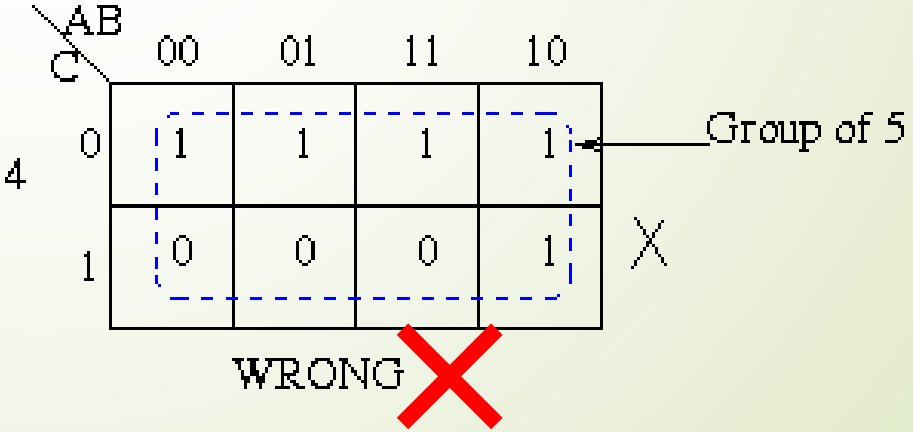
K-Map Rules (Cont.)

- Groups should have 2^n 1's. (n >= 0)



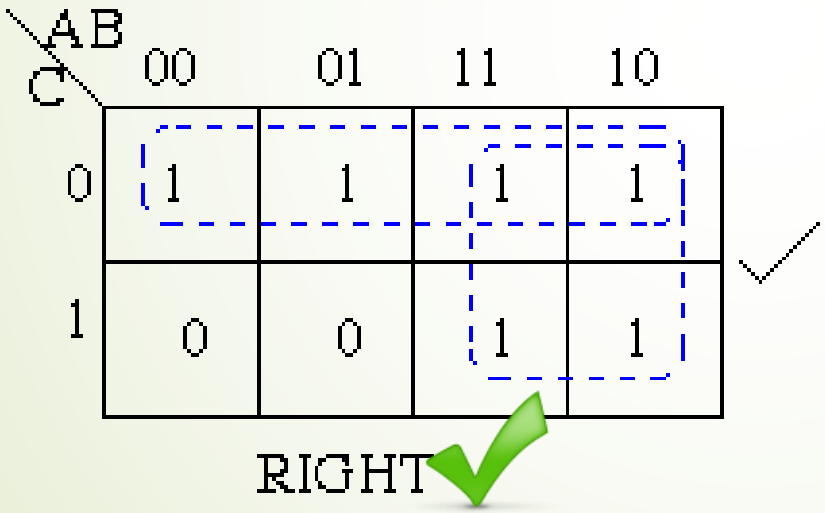


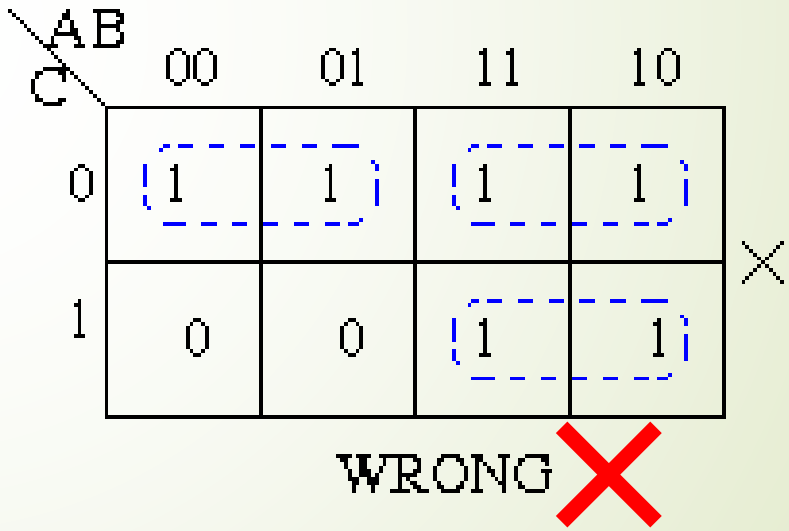




K-Map Rules (Cont.)

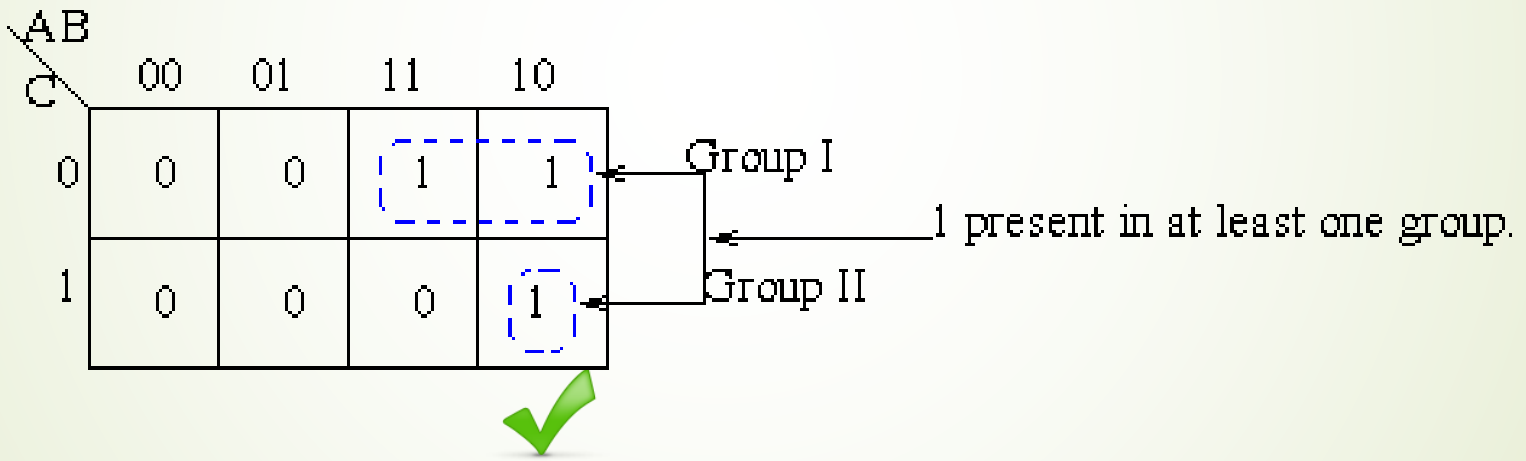
- Groups should grow up as big as possible considering **2^n rule**





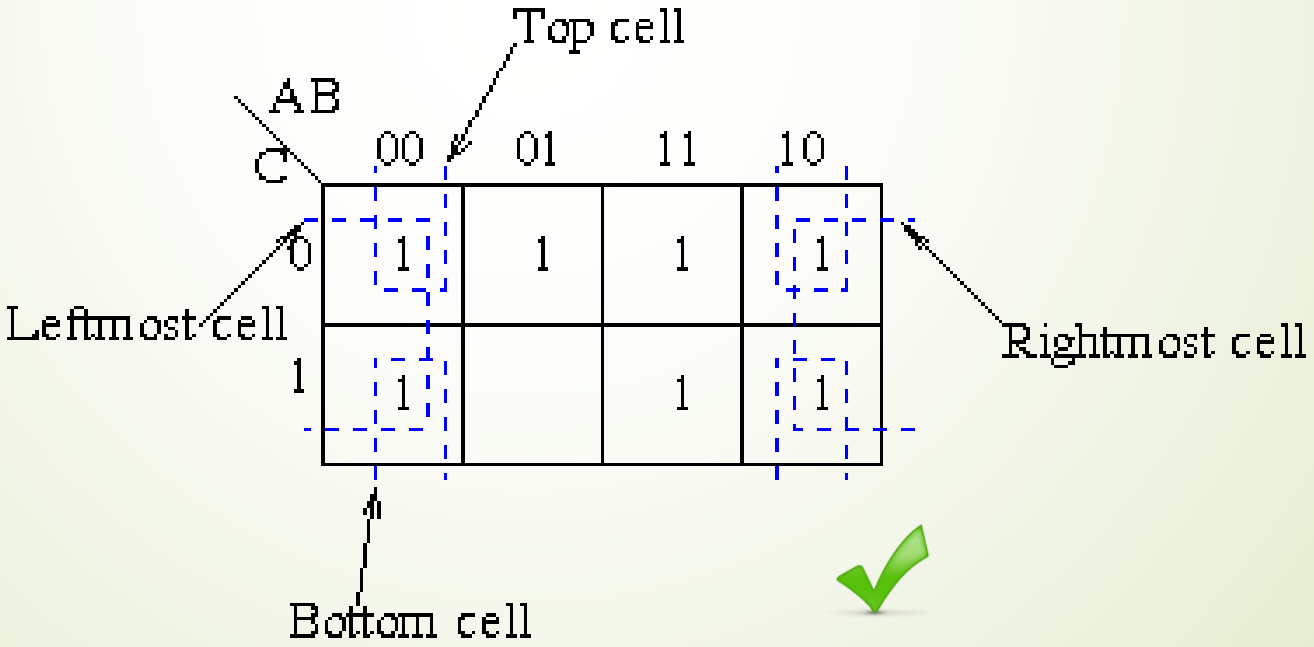
K-Map Rules (Cont.)

- Every 1 must present in at least one group



K-Map Rules (Cont.)

- Groups can be concatenated from top, bottom, rightmost and leftmost cells which satisfies one bit **Gray code** changes.



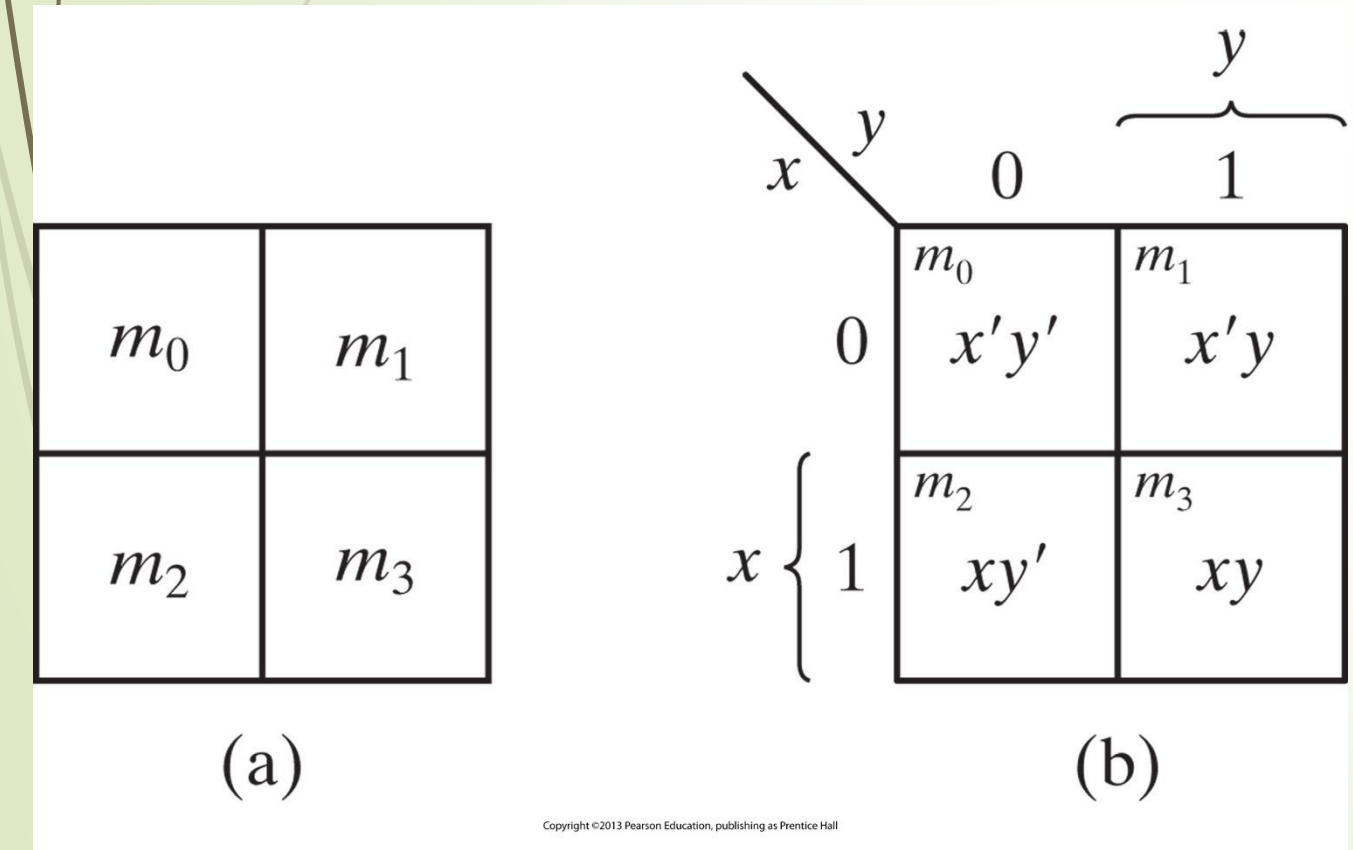
Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.
- Groups can overlap and wrap around the sides of the Kmap.

Two-Variable Map

- **Two-variable** has **four minterms**, and consists of four squares.
- $m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$



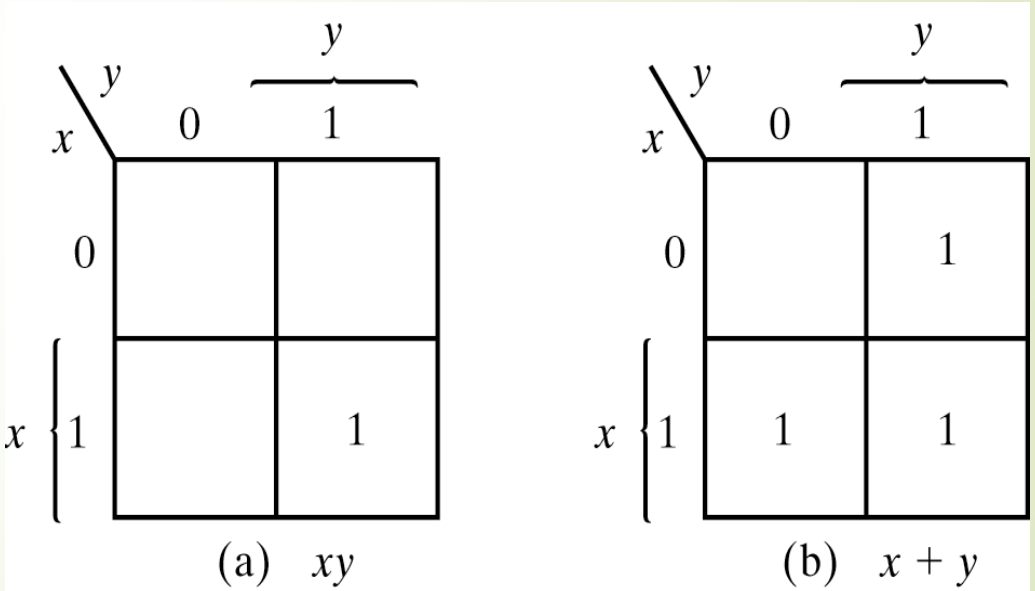
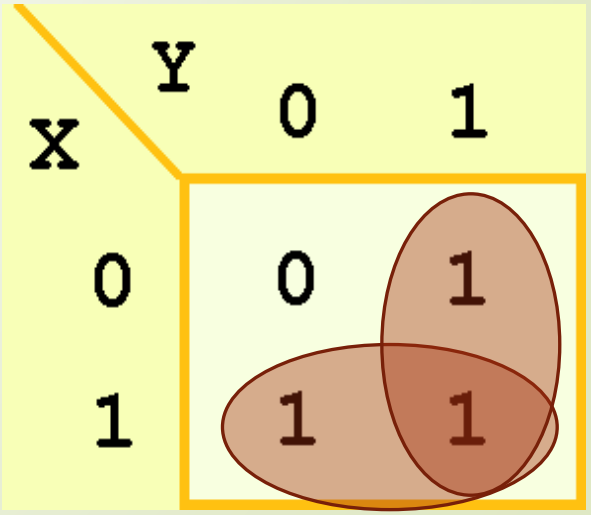


Fig. 3-2 Representation of Functions in the Map

Kmap Simplification for Two Variables

- Of course, the minterm function that we derived from our Kmap was not in simplest terms.
  - That's what we started with in this example.
- We can, however, reduce our complicated expression to its simplest terms by finding adjacent 1s in the Kmap that can be collected into groups that are powers of two.
- In our example, we have two such groups.

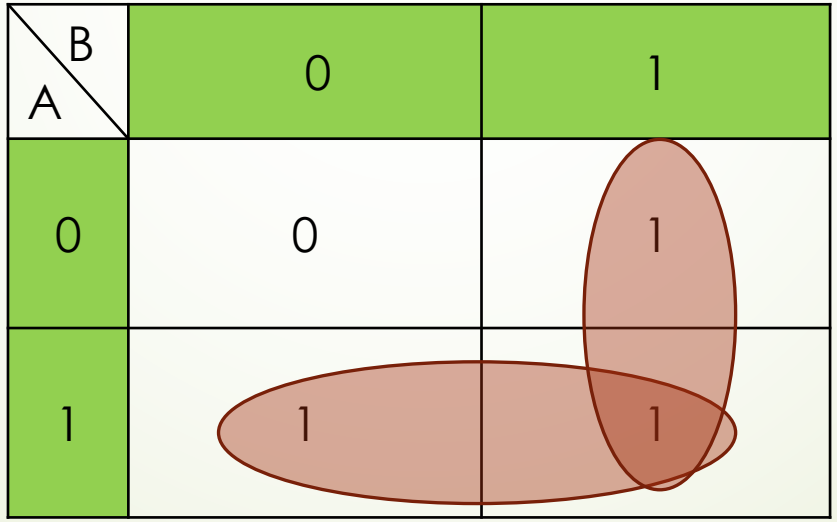


Two-Variable Map

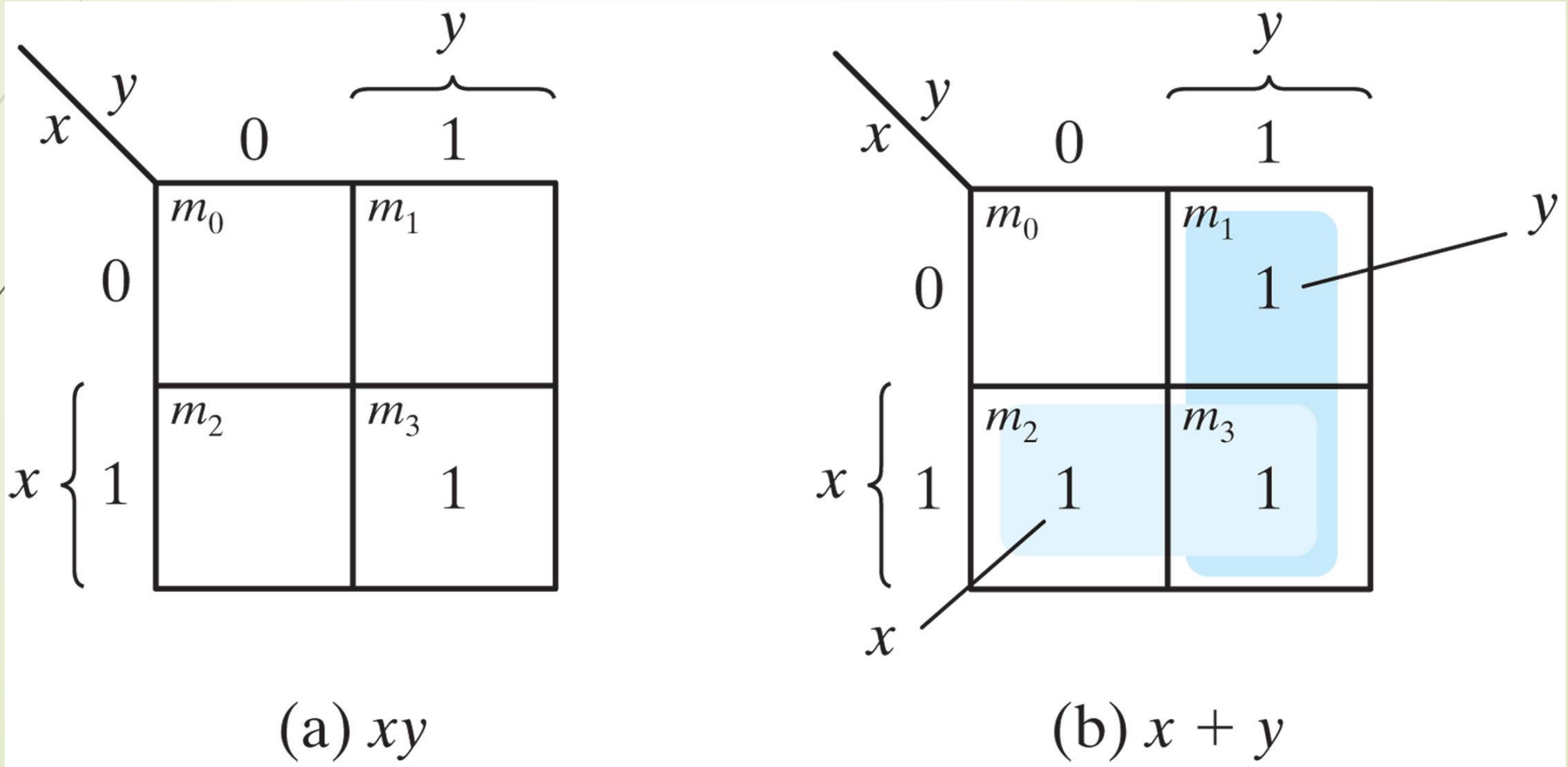
- OR Example
- Gray code is used
- F(A,B)=A'B+AB'+AB=A+B

Truth Table
S=A+B

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

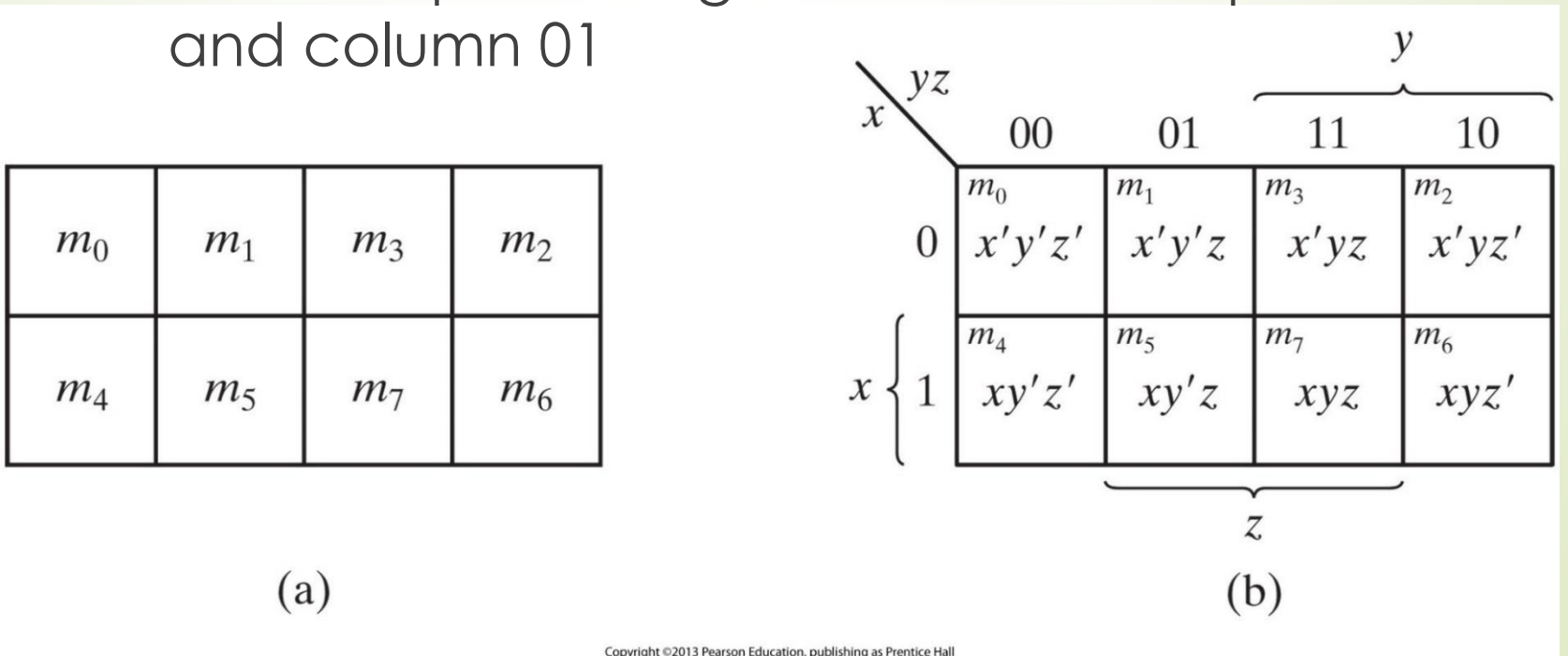


Representations of Functions in the Map



Three-Variable Map

- Note that the minterms are not arranged in a binary sequence, but similar to the Gray code.
  - **only one bit changes in value from one adjacent column to the next**
  - Ex: The square assigned to *m5* corresponds to row 1 and column 01



Three-Variable Map

- Thus, the first row of the K-map contains all minterms where x has a value of zero.
- The first column contains all minterms where y and z both have a value of zero.

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)

x \ yz	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

(y spans 11 and 10; z spans 01 and 11)

(b)

Three-Variable Map

- For simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares.
- **Any two adjacent squares in the map differ by only one variable**
- **The sum of two minterms** in **adjacent squares** can be simplified to a **single product term** consisting of **only two literals**.
  - EX:
    - $m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$
- Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable

Kmap Simplification for Three Variables

- Now for a more complicated Kmap. Consider the function:

$$F(X,Y,Z) = \bar{X}\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + \bar{X}YZ + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z}$$

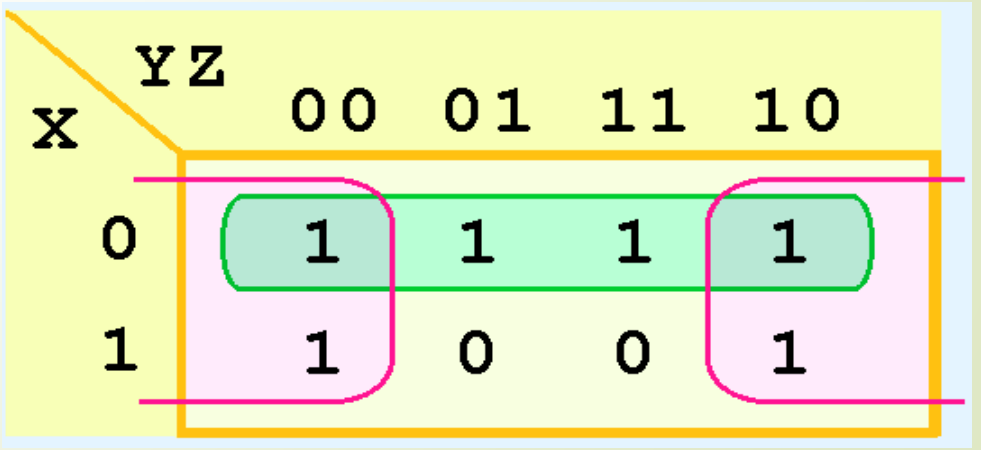
- Its Kmap is shown below. There are (only) two groupings of 1s.
  - Can you find them?

X \ YZ	00	01	11	10
0	1	1	1	1
1	1	0	0	1

Kmap Simplification for Three Variables

- In this Kmap, we see an example of a group that wraps around the sides of a Kmap.
- This group tells us that the values of x and y are not relevant to the term of the function that is encompassed by the group.
  - What does this tell us about this term of the function?

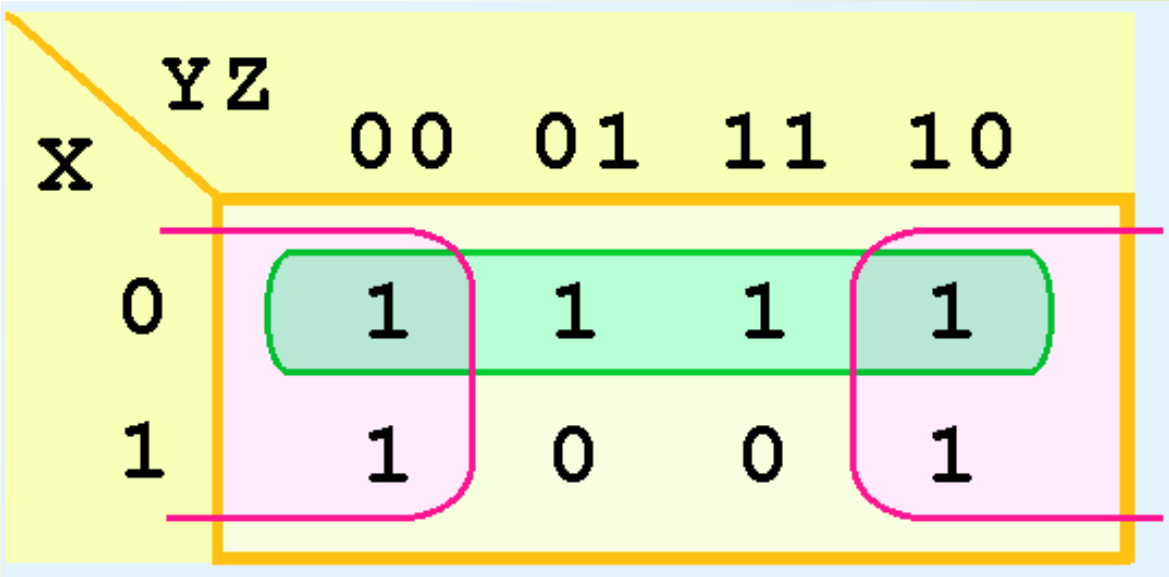
What about the green group in the top row?



Kmap Simplification for Three Variables

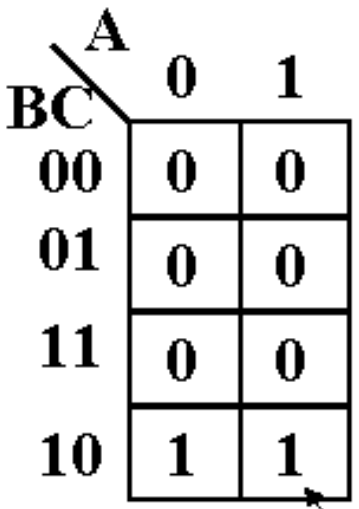
- The green group in the top row tells us that only the value of x is significant in that group.
- We see that it is complemented in that row, so the other term of the reduced function is $\overline{X}$.
- Our reduced function is: $F(X,Y,Z) = \overline{X} + \overline{Z}$

Recall that we had six minterms in our original function!



Three-Variable Map

Row	A B C	F(A,B,C)
0	0 0 0	0
1	0 0 1	0
2	0 1 0	1
3	0 1 1	0
4	1 0 0	0
5	1 0 1	0
6	1 1 0	1
7	1 1 1	0



$F(A,B,C) = \Sigma\, m(2,6)$

$= A'BC' + ABC'$

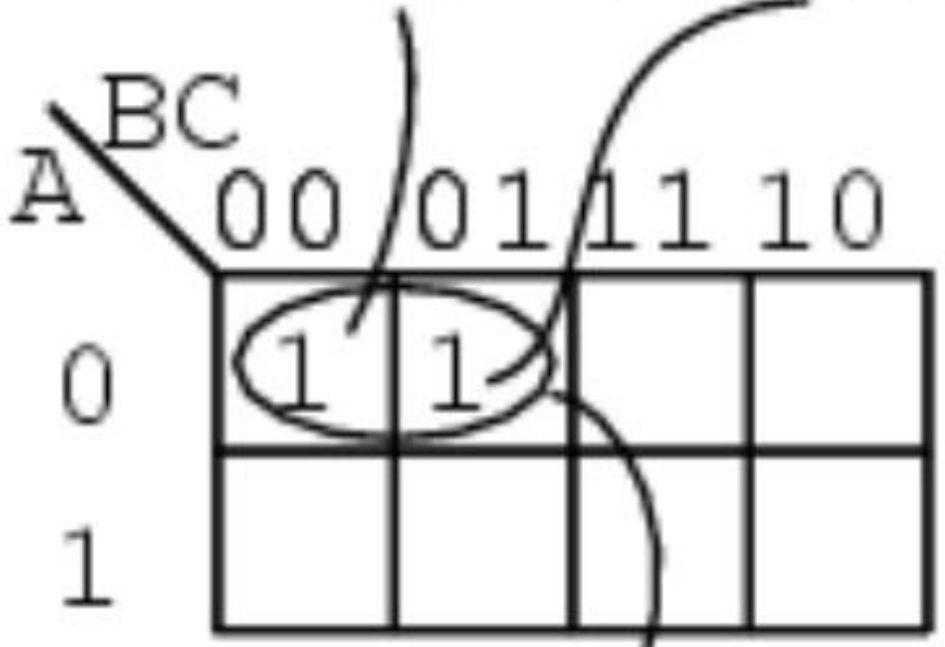
$= BC'(A'+A)$

$= BC'$

Boolean adjacency can be used to minimize functions!

K-Map Example #1 with three -variable

F(A,B,C)= $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,C$



F(A,B,C)= $\overline{A}\,\overline{B}$

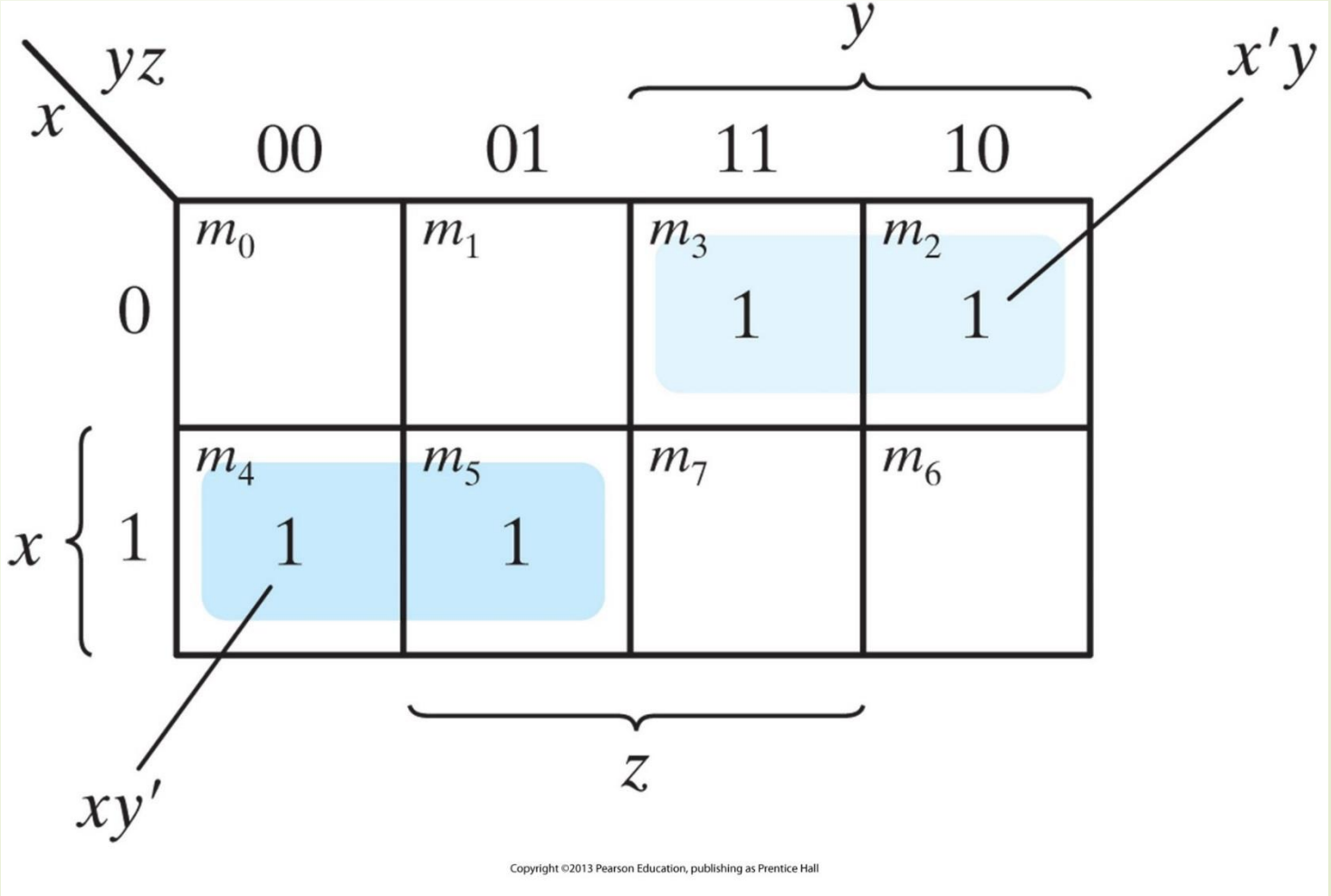
Example 3.1 Three-Variable Map

EXAMPLE 3.1

Simplify the Boolean function

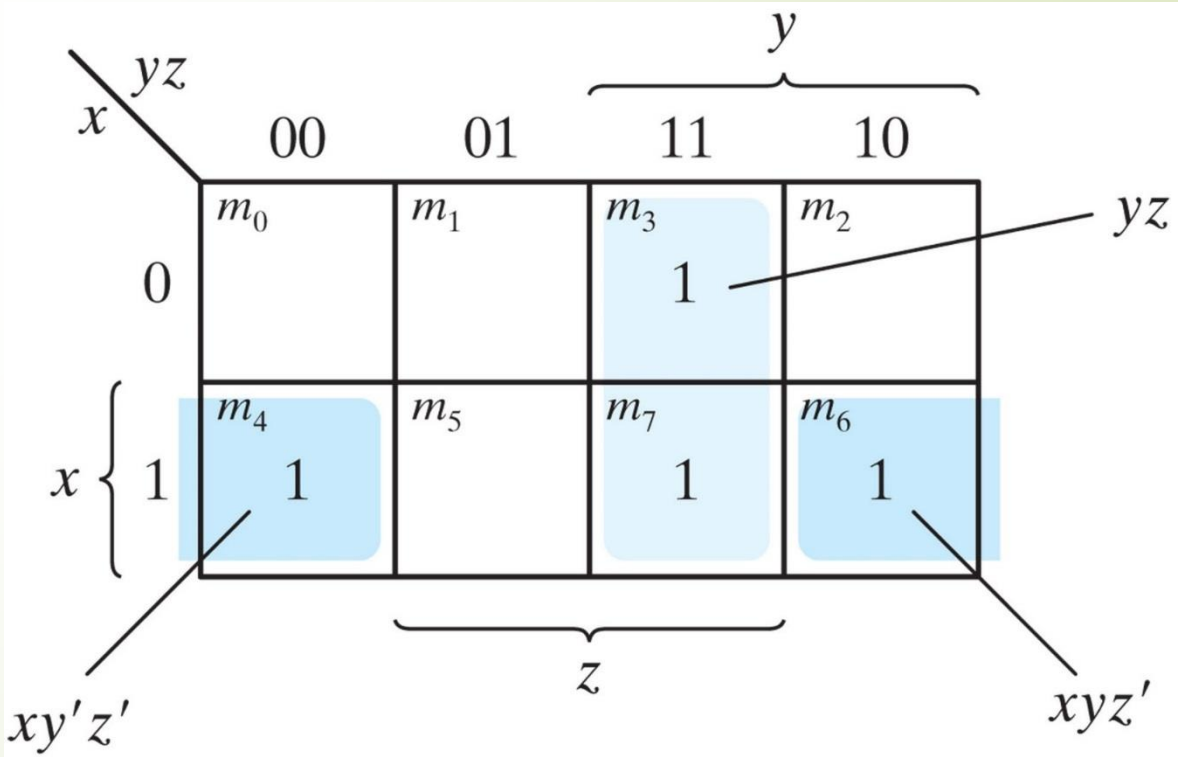
$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

Map for Example 3.1, $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$



Example 3.2 $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

- In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other



Combination of four adjacent squares in the three-variable map

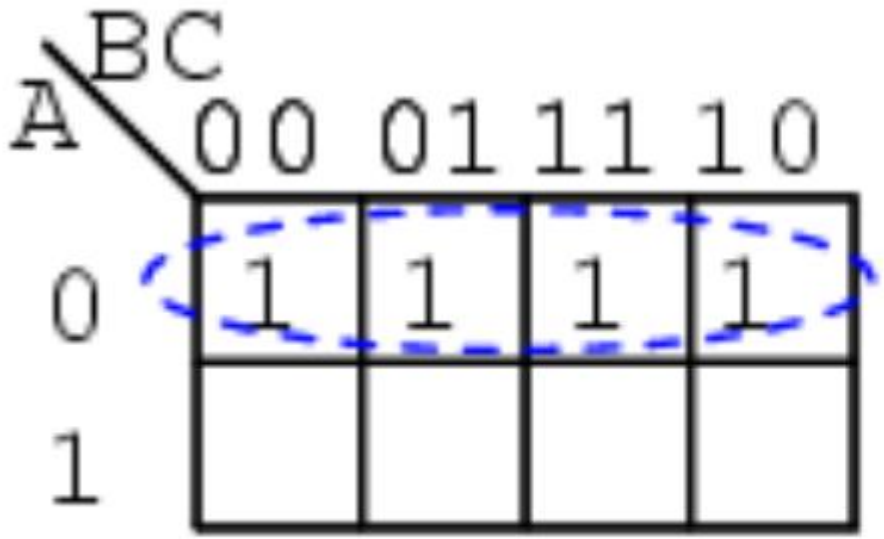
- Any such combination represents the logical sum of four minterms and results in an expression with **only one literal**
- As an example, the logical sum of the four adjacent minterms
  - 0, 2, 4, and 6 reduces to the single literal term z':
  - $m_0 + m_2 + m_4 + m_6 = x'y'z' + x'yz' + xy'z' + xyz'$

$$= x'z'(y' + y) + xz'(y' + y)$$

$$= x'z' + xz' = z'(x' + x) = z'$$

K-Map Example #2 with three -variable

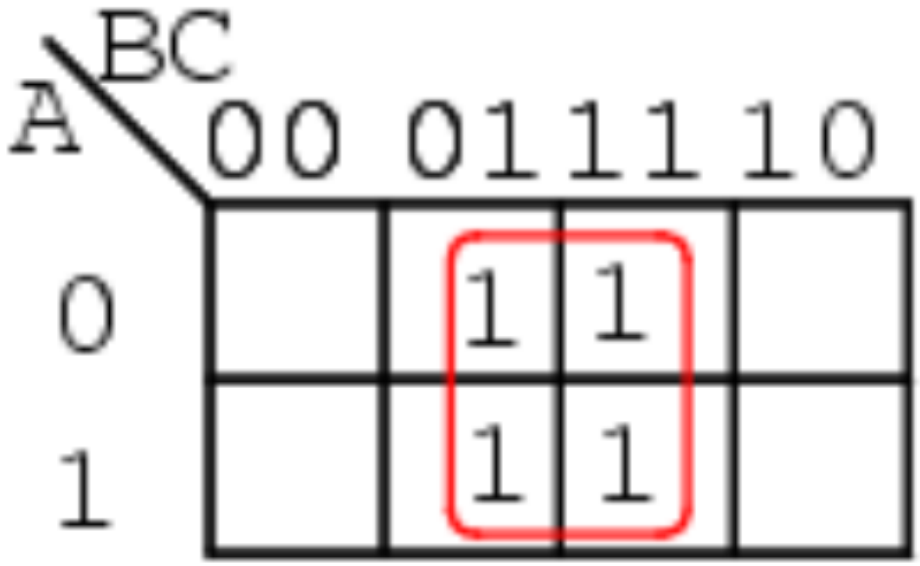
$$F(A,B,C)= \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}BC + \overline{A}\,B\overline{C}$$



$$F(A,B,C)= \overline{A}$$

K-map Example #3 with three -variable

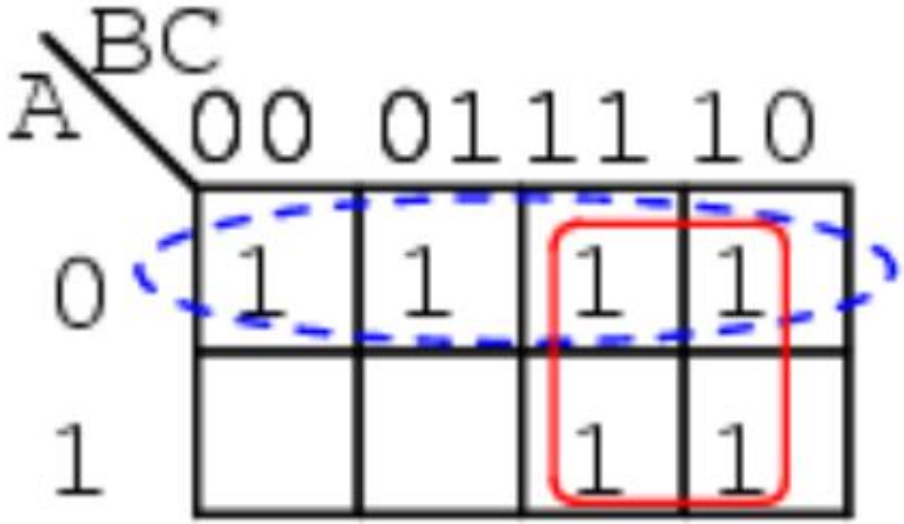
$$F(A,B,C) = \overline{A}\,\overline{B}C + \overline{A}BC + A\overline{B}C + ABC$$



$$F(A,B,C) = C$$

K-map Example #4 with three -variable

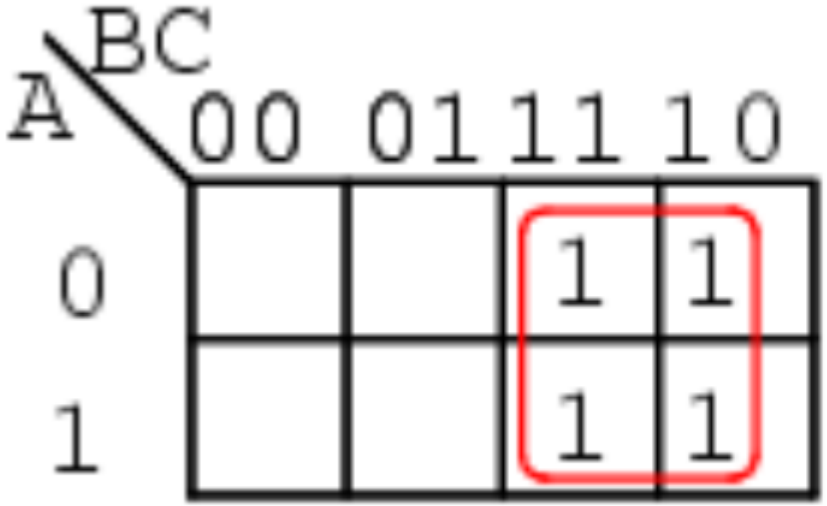
$$F(A,B,C) = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,B\,\overline{C} + ABC + A\,B\,\overline{C}$$



$$F(A,B,C) = \overline{A} + B$$

K-map Example #5 with three -variable

$$F(A,B,C) = \overline{A}BC + \overline{A}B\overline{C} + ABC + AB\overline{C}$$



$$F(A,B,C) = B$$

K-map Example #6 with three -variable

A \ BC	00	01	11	10
0		1	1	
1		1	1	1

F (A, B, C) = C + AB

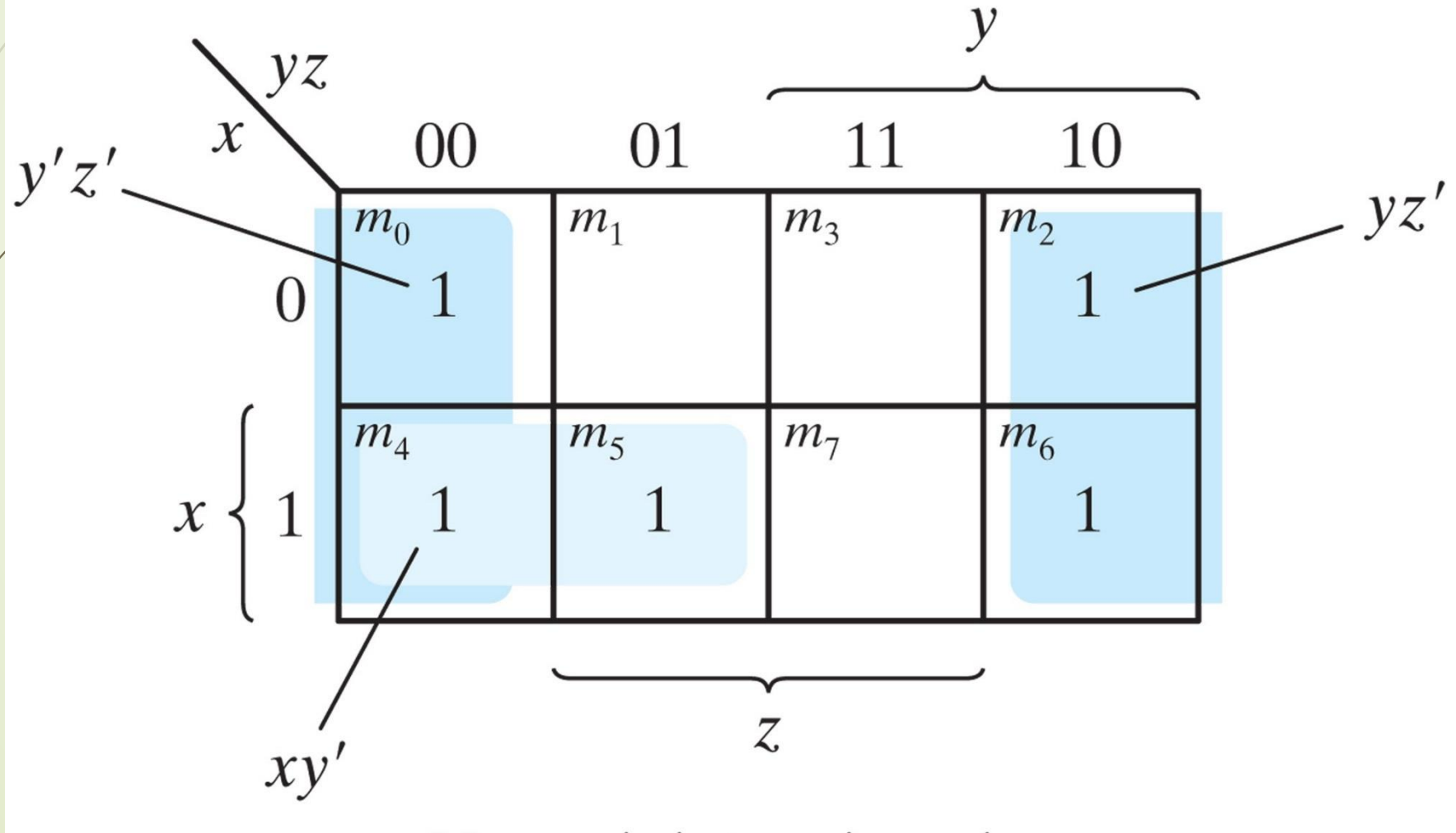
Example 3.3

EXAMPLE 3.3

Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

Map for Example 3.3, $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$



Missing input literals

- If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms

Example 3.4

EXAMPLE 3.4

For the Boolean function

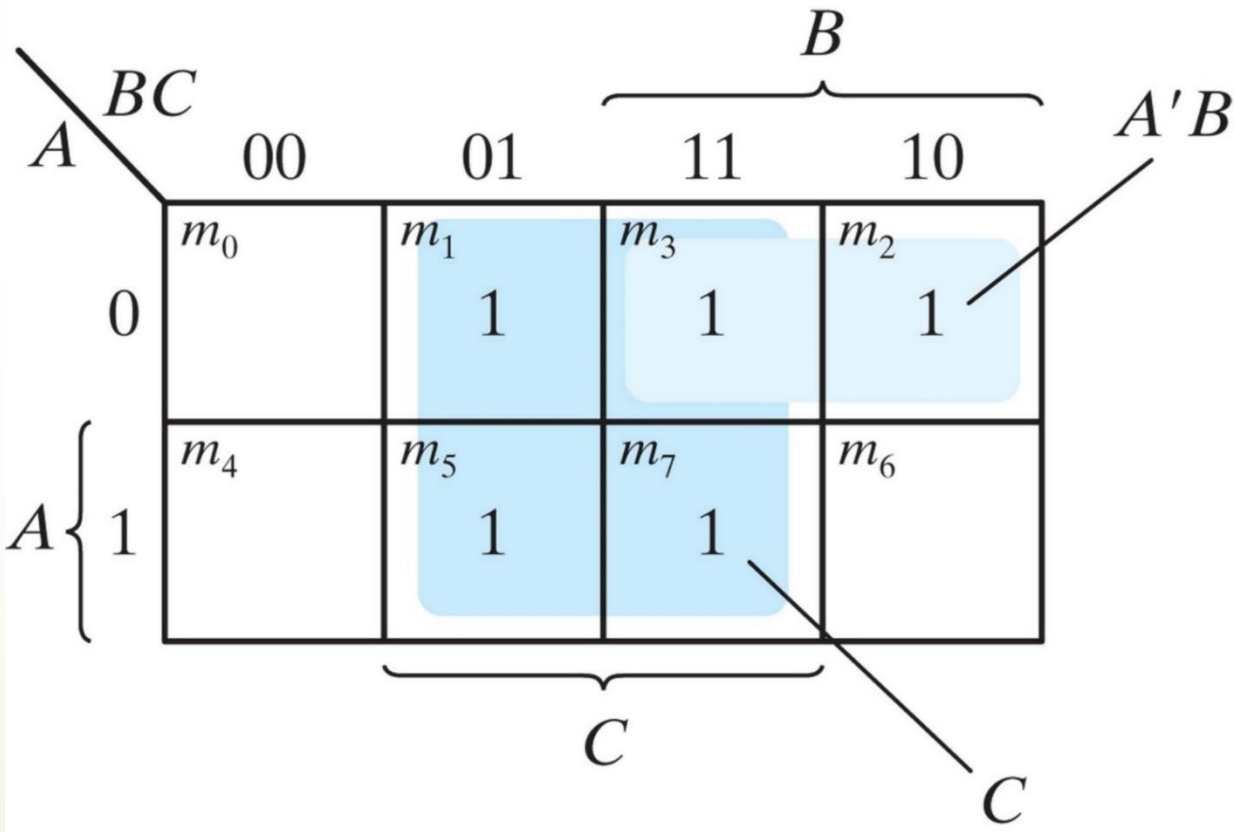
$$F = A'C + A'B + AB'C + BC$$

(a) Express this function as a sum of minterms.

(b) Find the minimal sum-of-products expression.

Map of Example 3.4, *A′C + A′B + AB′C + BC = C + A′B*

- *F* (*A*, *B*, *C*) = ∑(1, 2, 3, 5, 7)
- The sum-of-products expression, as originally given, has too many Terms
- *F = C + A'B*



Four-Variable K-map

- The numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13
- Thus, the square in the third row and second column represents minterm m_{13}

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

(a)

wx \ yz	00	01	11	10
00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$
01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$
11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$
10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$

y (over 11, 10); x (right, over 01, 11, 10); w (left, over 11, 10); z (below 00, 01, 11, 10)

(b)

Four-Variable K-map

- 1 square = 1 minterm = 4 literals
- 2 adjacent squares = 1 term = 3 literals
- 4 adjacent squares = 1 term = 2 literals
- 8 adjacent squares = 1 term = 1 literal
- 16 adjacent squares = 1

Example 3.5 Four-Variable K-map

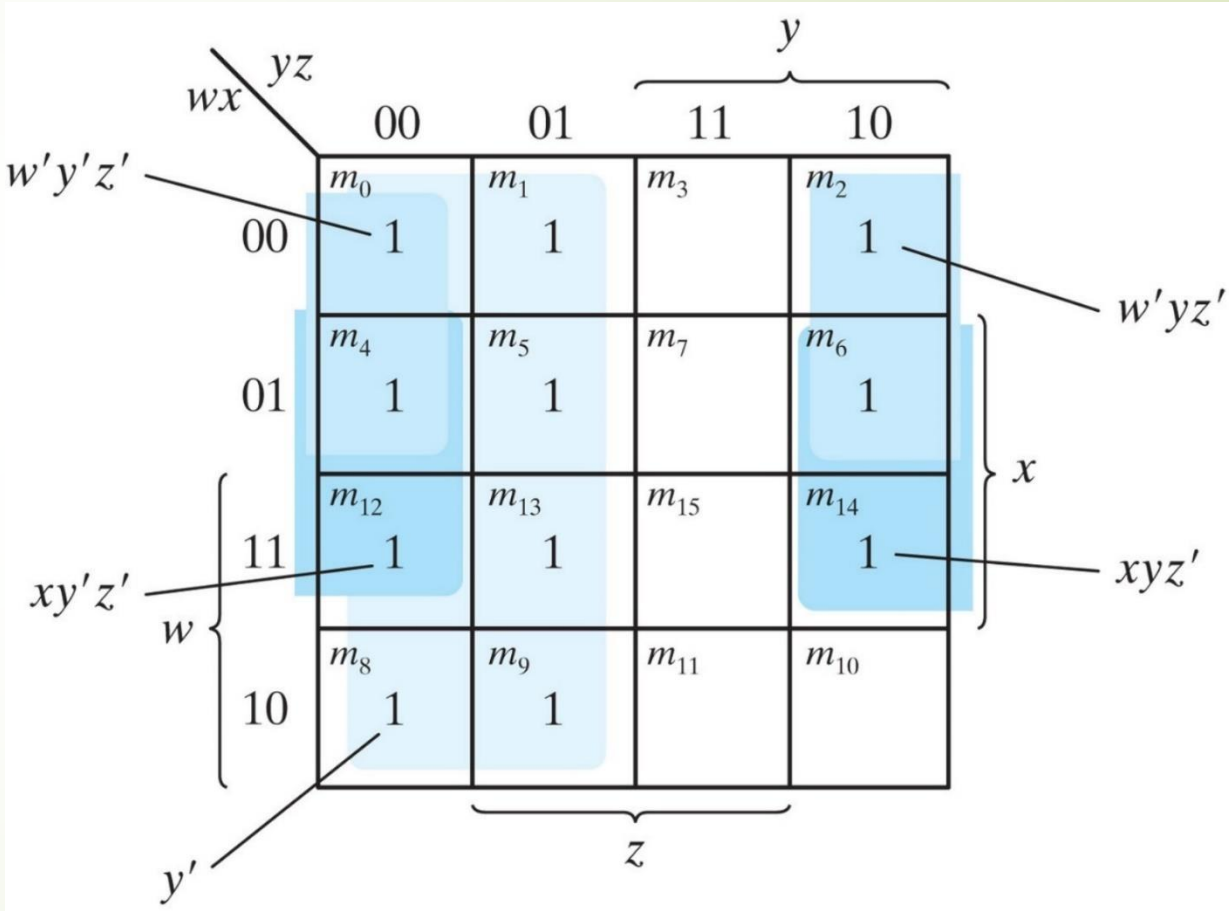
EXAMPLE 3.5

Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Example 3.5

- $F(w, x, y, z)$
  $= \Sigma(0,1,2,4,5,6,8,9,12,13,14)$
  $= y' + w'z' + xz'$



Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$

Example 3.6

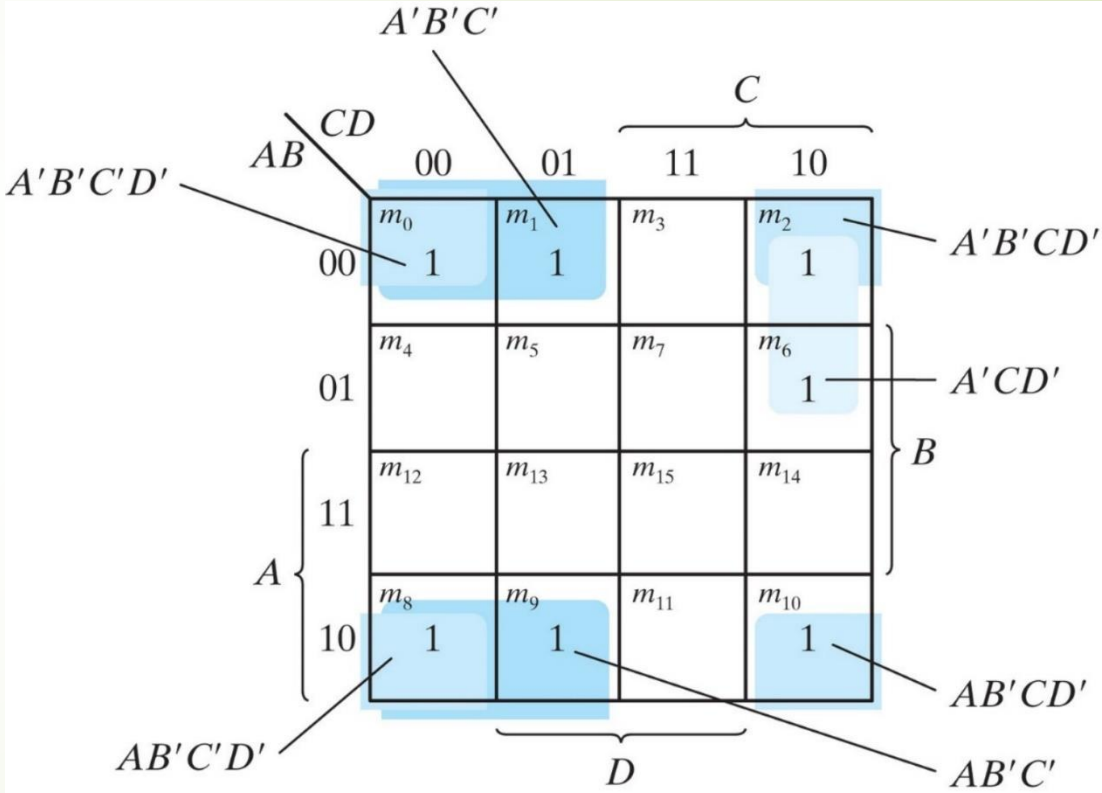
EXAMPLE 3.6

Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

Example 3.6

- $A'B'C' + B'CD' + A'BCD' + AB'C'$
  $= B'D' + B'C' + A'CD'$



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$

K-map Simplification for Four Variables

- We have populated the Kmap shown below with the nonzero minterms from the function:

$$F(W,X,Y,Z) = \bar{W}\bar{X}\bar{Y}\bar{Z} + \bar{W}\bar{X}\bar{Y}Z + \bar{W}\bar{X}Y\bar{Z} + \bar{W}X\bar{Y}\bar{Z} + W\bar{X}\bar{Y}\bar{Z} + W\bar{X}\bar{Y}Z + W\bar{X}Y\bar{Z}$$

- Can you identify (only) three groups in this Kmap?

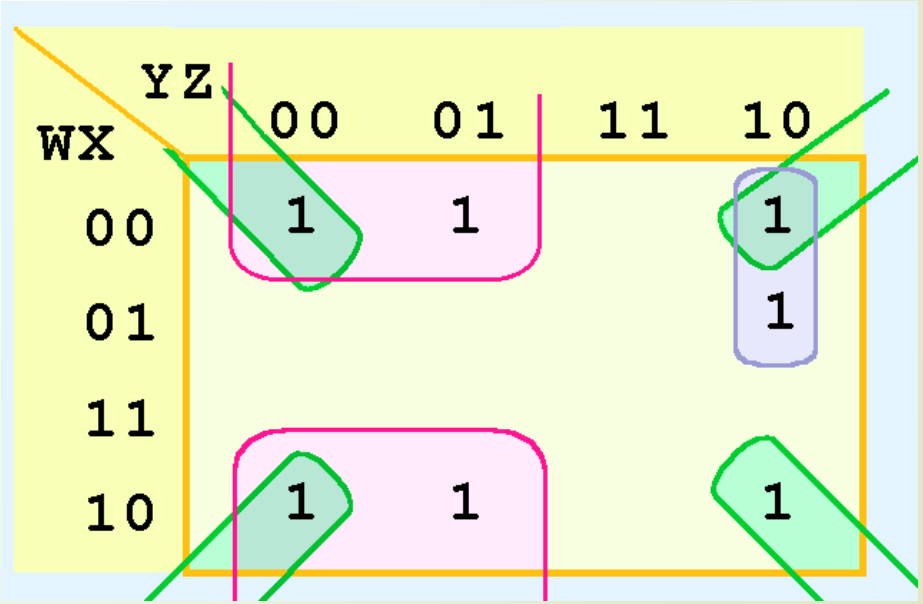
Recall that groups can overlap.

WX \ YZ	00	01	11	10
00	1	1		1
01				1
11				
10	1	1		1

Kmap Simplification for Four Variables

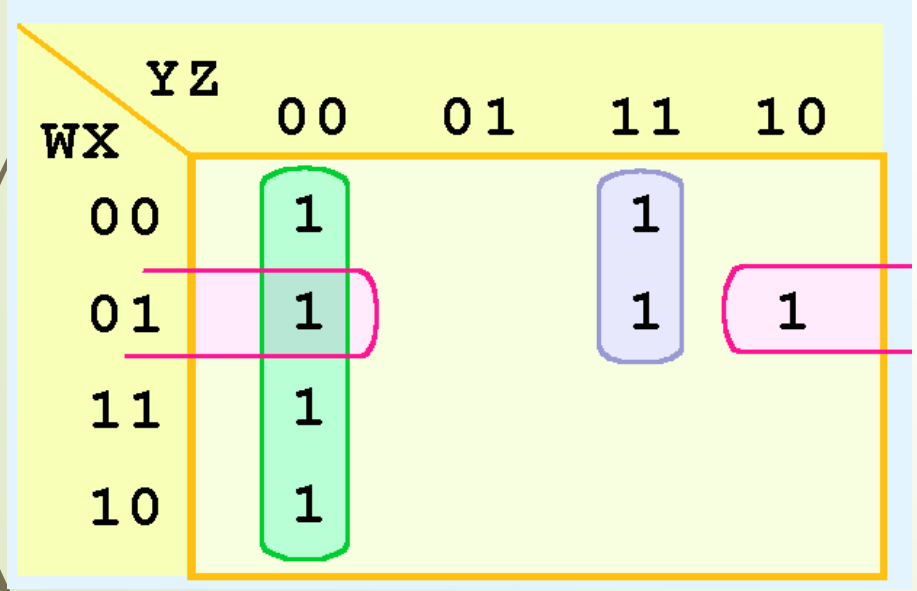
- Our three groups consist of:
  - A purple group entirely within the Kmap at the right.
  - A pink group that wraps the top and bottom.
  - A green group that spans the corners.
- Thus we have three terms in our final function:

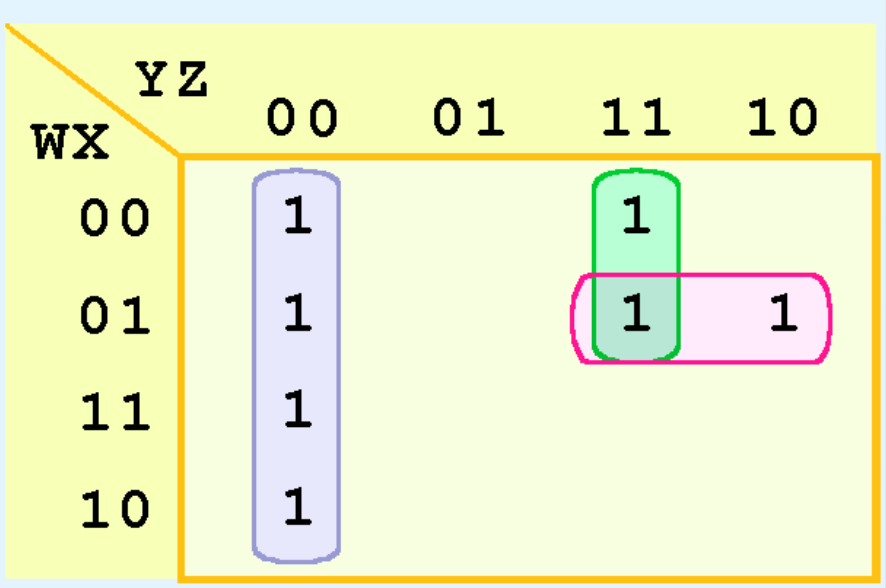
$$F(W,X,Y,Z) = \bar{W}\bar{Y} + \bar{X}\bar{Z} + \bar{W}Y\bar{Z}$$



Kmap Simplification for Four Variables

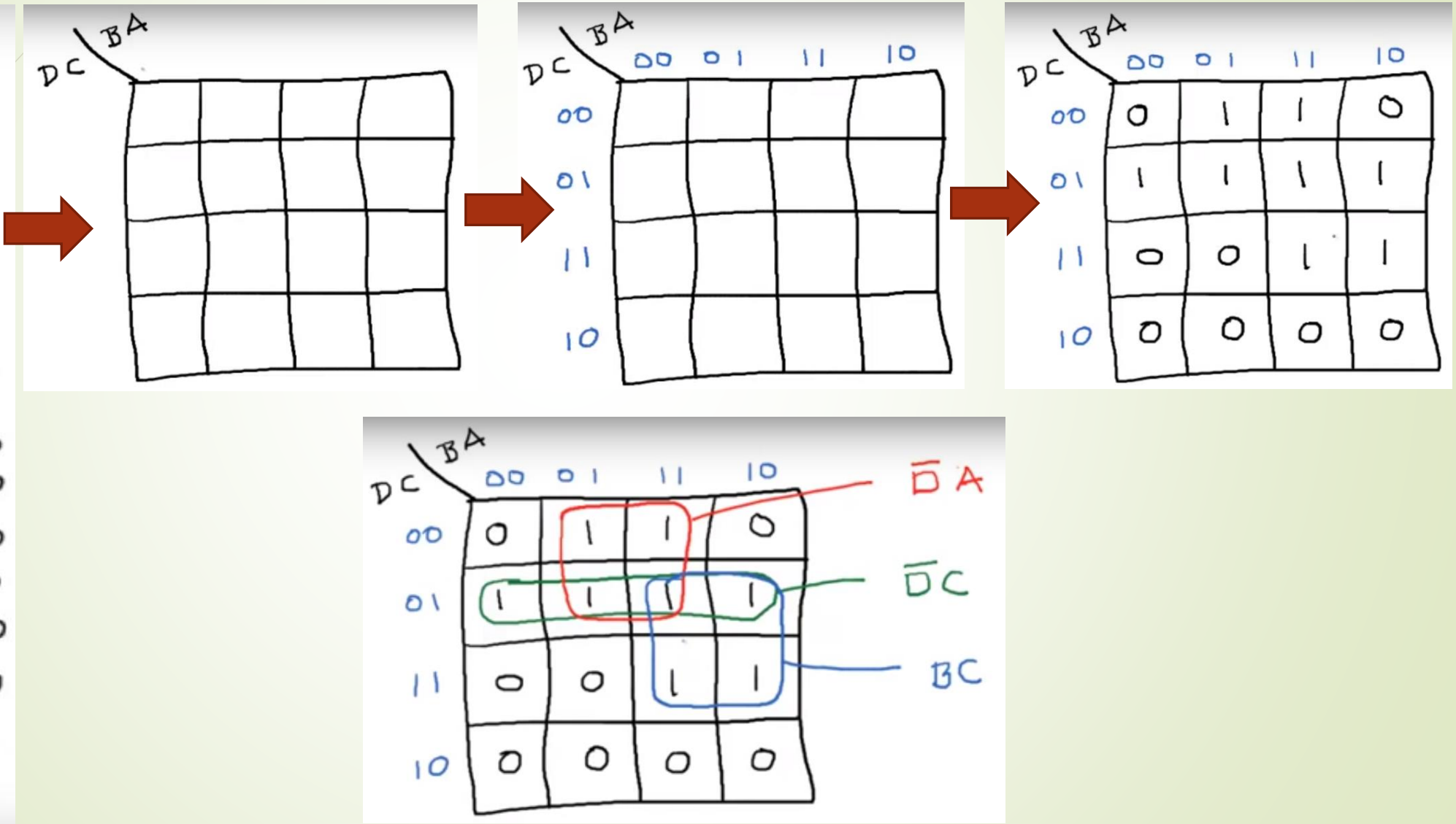
- It is possible to have a choice as to how to pick groups within a Kmap, while keeping the groups as large as possible.
- The (different) functions that result from the groupings below are **logically equivalent**.





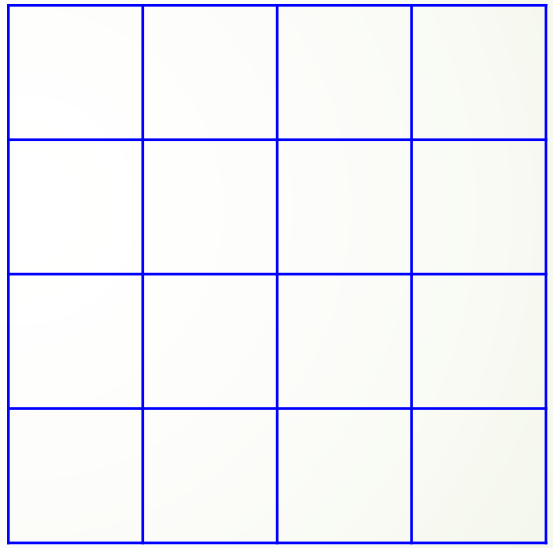
Kmap Simplification for Four Variables

D	C	B	A	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1



Kmap Simplification for Four Variables

R	S	T	U	F_3
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**0**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**1**



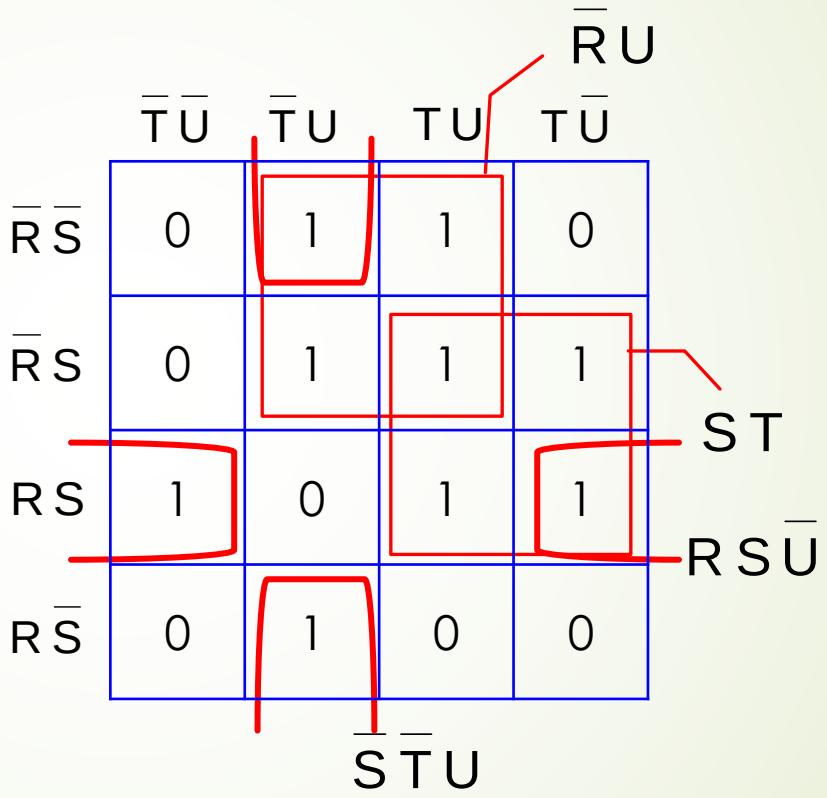
Kmap Simplification for Four Variables

R	S	T	U	F_3
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**0**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**1**

	$\bar{T}\bar{U}$	$\bar{T}U$	TU	$T\bar{U}$
$\bar{R}\bar{S}$	0	1	1	0
$\bar{R}S$	0	1	1	1
RS	1	0	1	1
$R\bar{S}$	0	1	0	0

Kmap Simplification for Four Variables

R	S	T	U	F_3
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**0**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**1**



$$F_3 = RS\bar{U} + \bar{S}\bar{T}U + \bar{R}U + ST$$

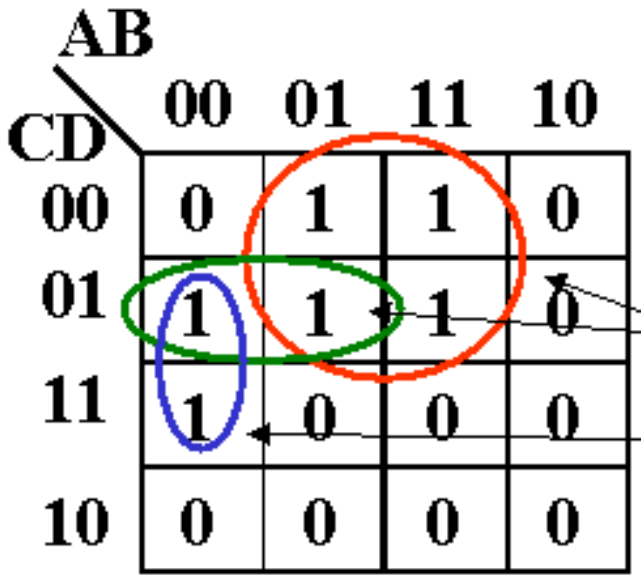
Prime implicants

- **A *prime implicant*** is a **product term** obtained by combining the **maximum possible number of adjacent squares** in the map.
- **The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares**
  - This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's.
- If a minterm in a square is covered by only one prime implicant, that the prime implicant is said to be essential.
- In choosing adjacent squares in a map, we must ensure that
  - (1) all the minterms of the function are **covered** when we combine the squares,
  - (2) the number of terms in the expression is **minimized**, and
  - (3) there are **no redundant terms** (i.e., minterms already covered by other terms).

Prime implicants

- Two adjacent 1's form a **prime implicant**, provided that they are not within a group of four adjacent squares
- Four adjacent 1's form a **prime implicant** if they are not within a group of eight adjacent squares, and so on
- Identification of the prime implicants in the map **helps in determining the alternatives** that are available for obtaining a simplified expression
- Each combination may produce an equally simplified expression.

Non-Essential vs Essential Prime implicants



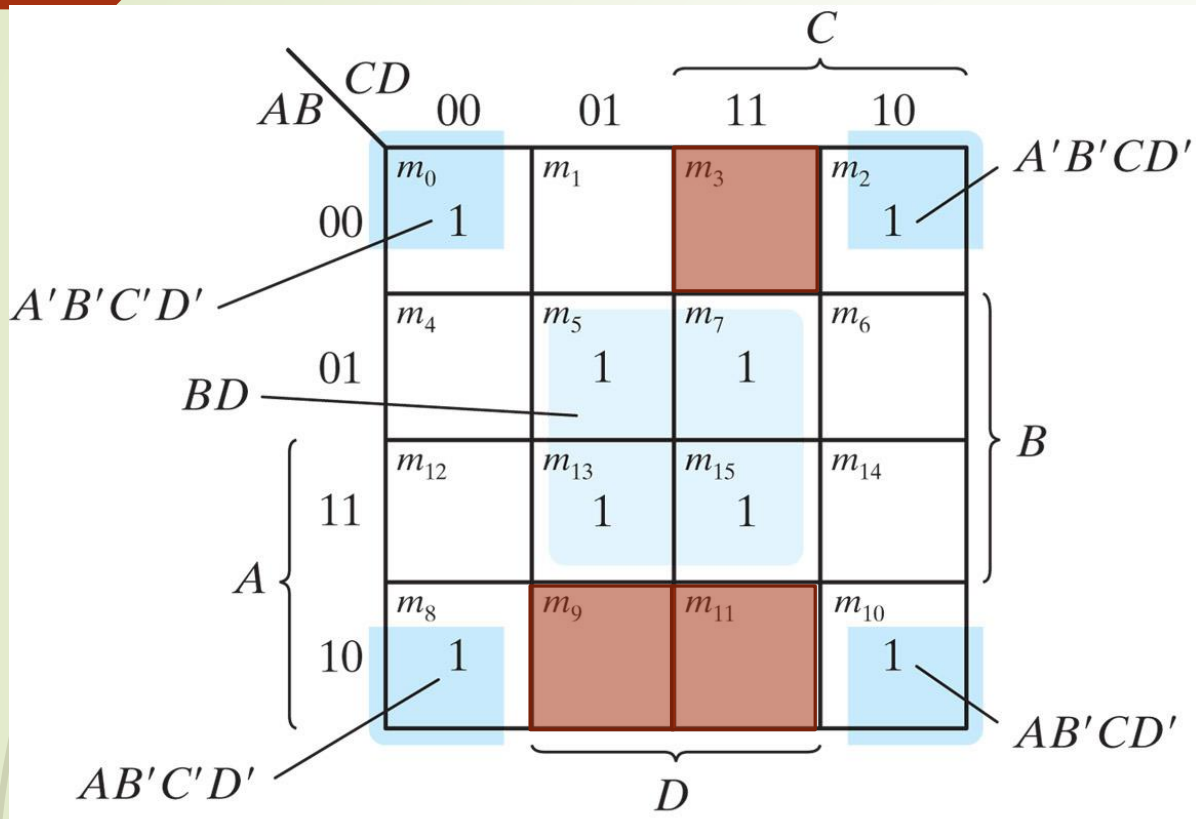
EACH of the coverings is a PRIME IMPLICANT.

BC' , A'C'D , A'B'D

F(A,B,C,D) = BC' + A'B'D (minimum # of PIs)

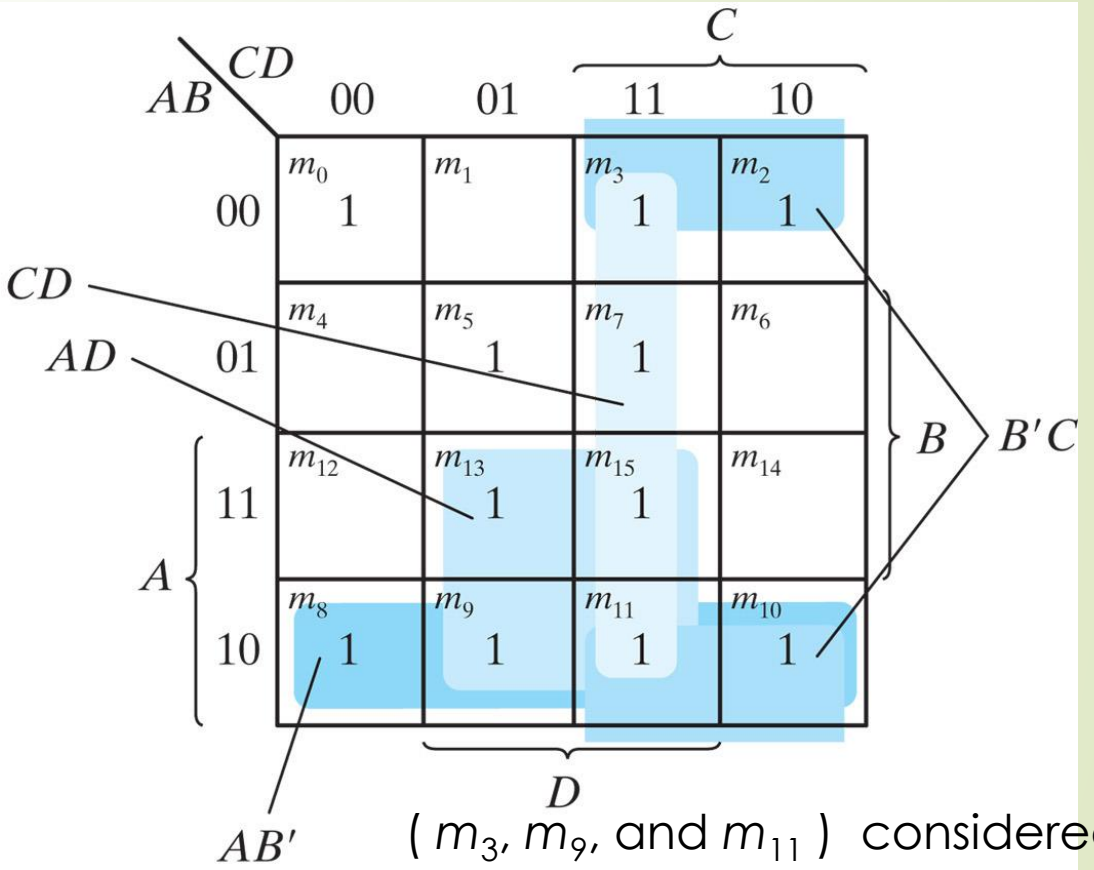
Prime Implicant A'C'D is a NON-ESSENTIAL prime implicant because its '1's are covered by other PIs . A PI is ESSENTIAL if it covers a MINTERM that cannot be covered by any other PI.

Prime implicants for $F(A, B, C, D) = \sum(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

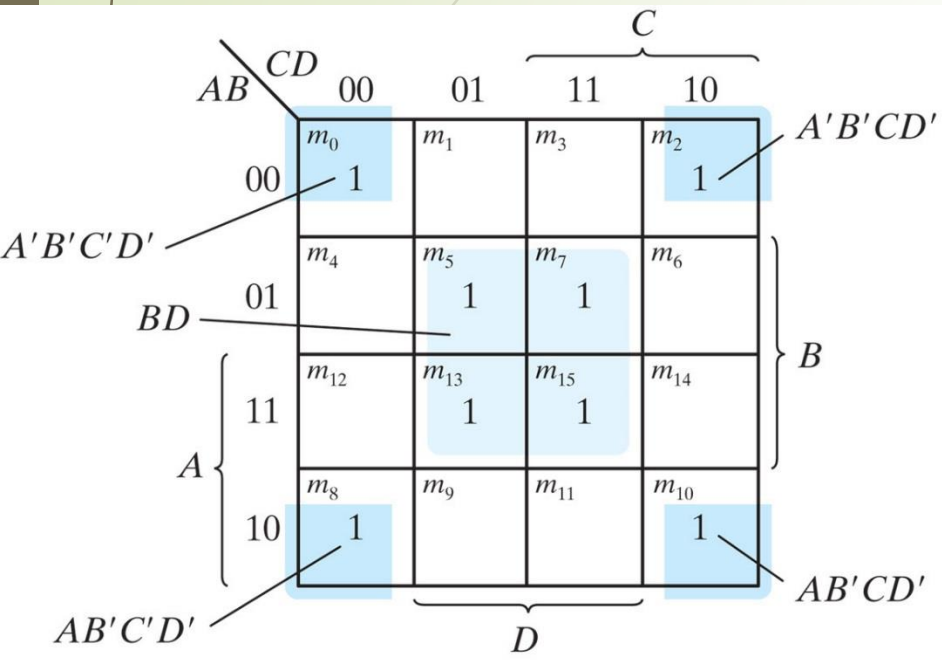
(a) Essential prime implicants BD and $B'D'$



(b) Prime implicants CD, $B'C$, AD, and AB'

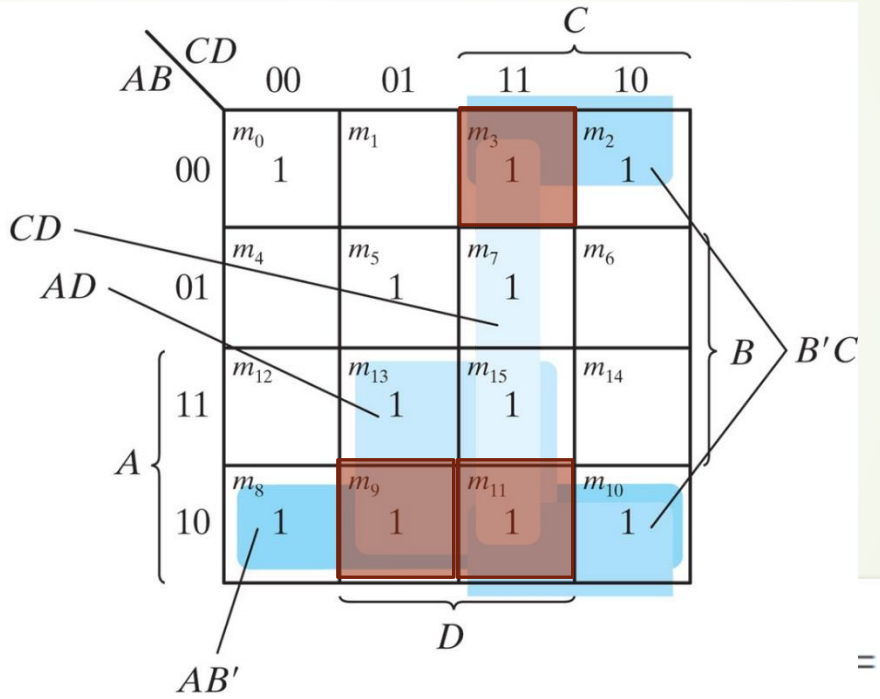
(m_3, m_9, and m_{11}) considered

Prime implicants for $F(A, B, C, D) = \sum(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and $B'D'$



(b) Prime implicants CD, $B'C$, AD, and AB'

- This shows all possible ways that the three minterms(m_3,m_9, m_{11}) can be covered with prime implicants.

$$= BD + B'D' + CD + AD$$
$$= BD + B'D' + CD + AB'$$
$$= BD + B'D' + B'C + AD$$
$$= BD + B'D' + B'C + AB'$$

Prime implicants (Cont.)

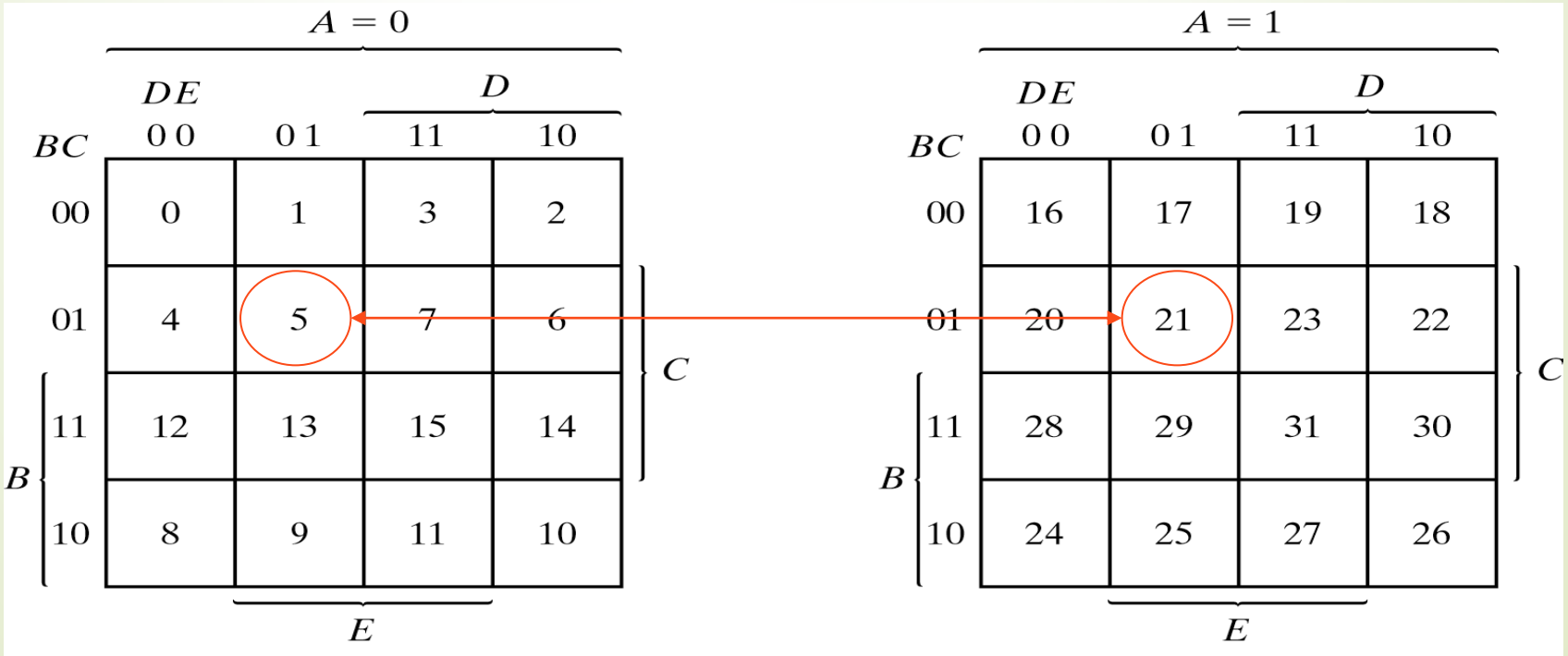
- The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants
- The simplified expression is obtained from the **logical sum of all the essential prime implicants**, **plus** other prime implicants that may be needed to cover any **remaining minterms not covered by the essential prime implicants**
- Each combination may produce an equally simplified expression.

Five and six-Variable Maps

- When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved
- Maps for more than four variables are difficult to use therefore we need to use software

Five and six-variable Maps

- The left-hand four-variable map represents the 16 squares where A=0, and the other four-variable map represents the squares where A=1.
- In addition, each square in the A=0 map is adjacent to the corresponding square in the A=1 map.



Six-variable k-map

- 64 cells
- Video

PRODUCT-OF-SUMS SIMPLIFICATION

- All of the previous examples were expressed in **sum-of-products form**
- The 1's placed in the squares of the map represent the minterms of the function
- The complement of a function is represented in the map by the squares not marked by 1's.
- If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of ***F'***).
- The complement of F' gives us back the function F in product-of-sums form (a consequence of DeMorgan's theorem).
- Because of the generalized DeMorgan's theorem, the function so obtained is automatically in **product-of-sums form**

Example 3.7 Products-of-sums

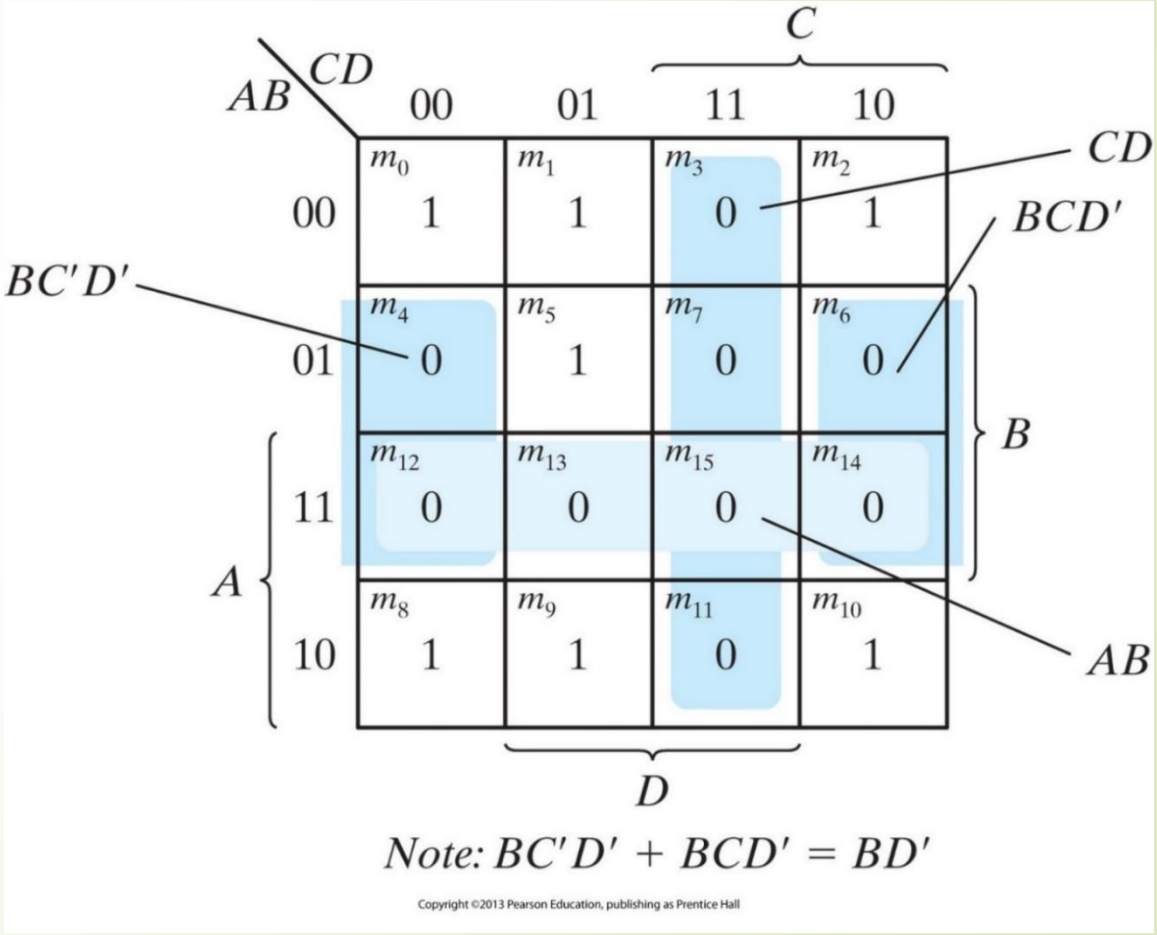
EXAMPLE 3.7

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

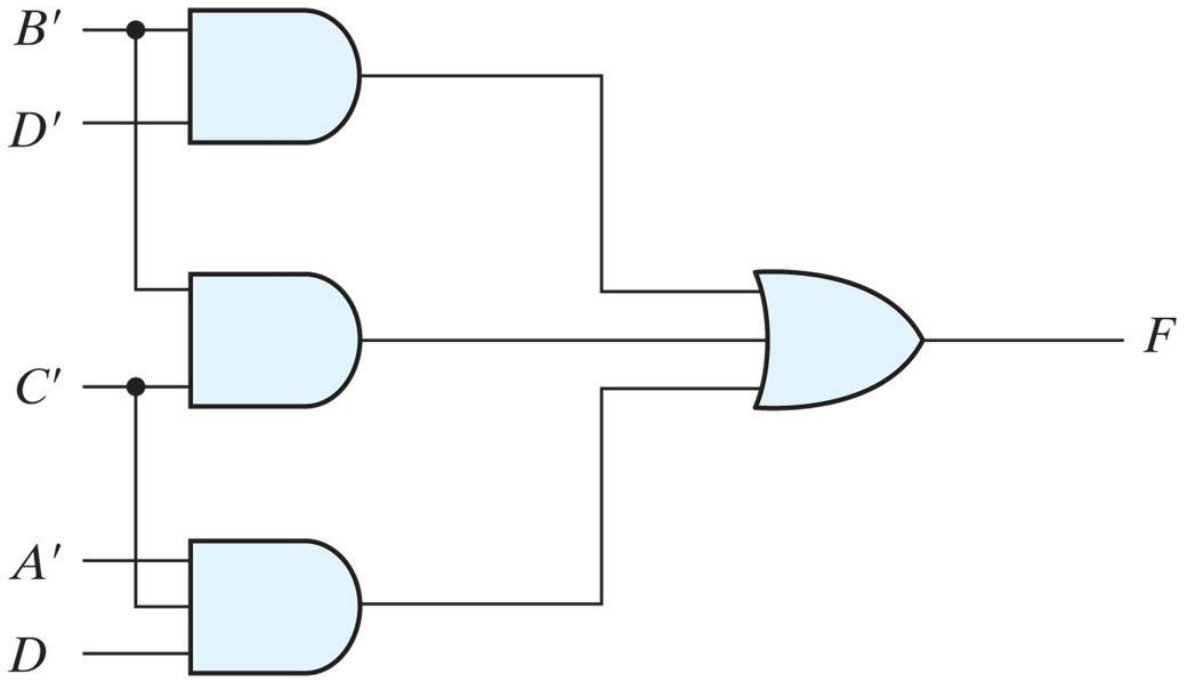
$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

Example 3.7 Products-of-sums simplification

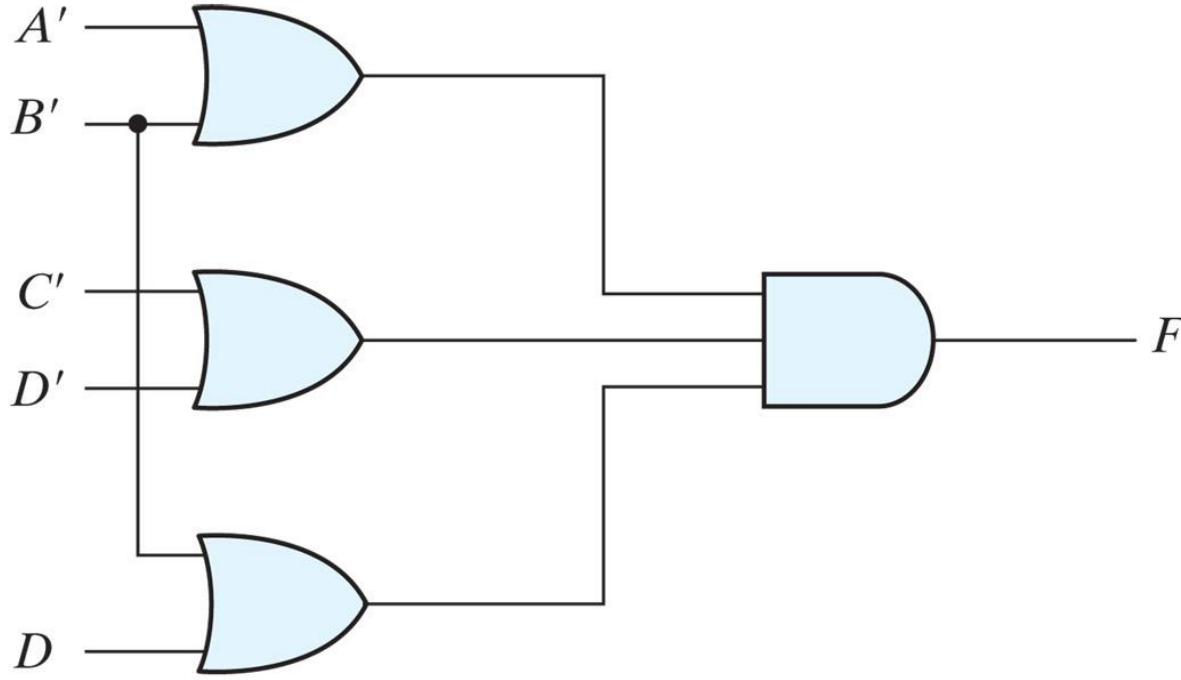
- $F(A, B, C, D) = \sum(0, 1, 2, 5, 8, 9, 10)$
- **Sum-of-Products form**
  - $F = B'D' + B'C' + A'C'D$
  - $F' = AB + CD + BD'$
- Applying DeMorgan's theorem
  - by taking the dual and
  - complementing each literal
  - we obtain the simplified function
- **In product-of-sums form:**
  - $F = (A' + B')\ (C' + D')\ (B' + D)$



Gate implementations of the function of Ex.3.7



(a) $F = B'D' + B'C' + A'C'D$



(b) $F = (A' + B')(C' + D')(B' + D)$

Products-of-sums simplification

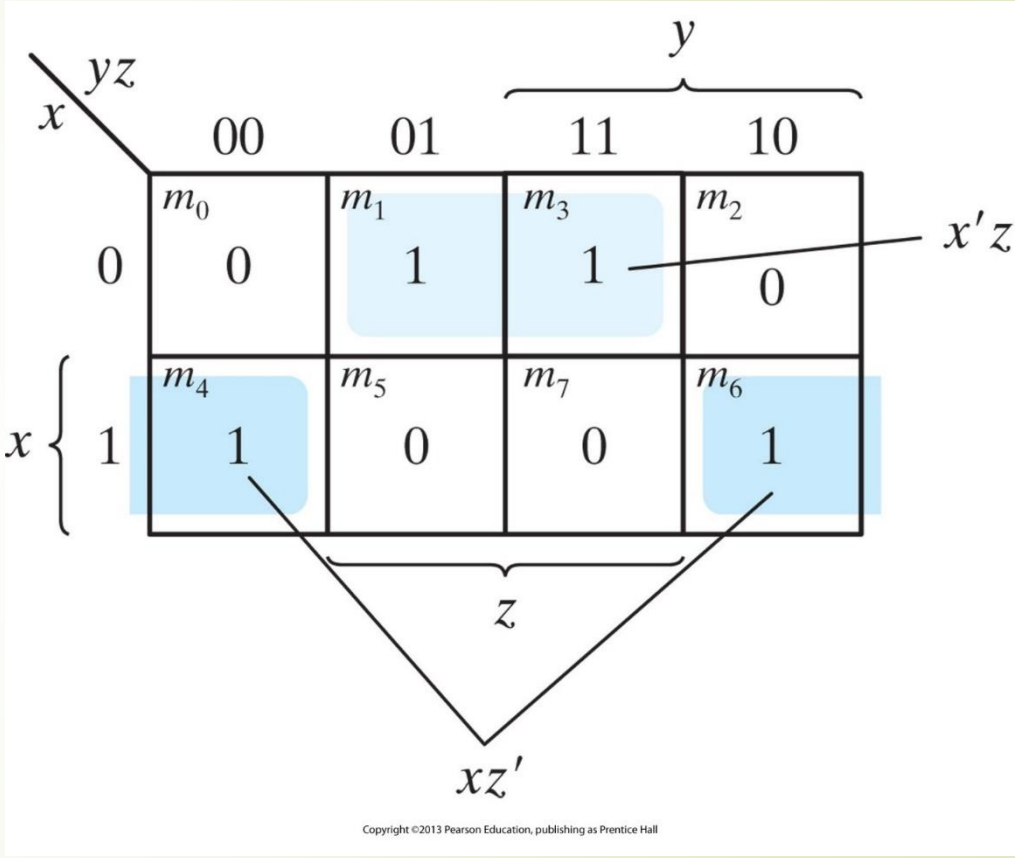
- Consider, for example, the truth table that defines the function *F* in Table 3.1
- In sum-of-minterms form, this function is expressed as:
  - $F(x, y, z) = \sum(1, 3, 4, 6)$
- In product-of-maxterms form, it is expressed as:
  - $F(x, y, z) = \prod(0, 2, 5, 7)$

Table 3.1
Truth Table of Function F

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Map for the function of Table 3.1

- For the **sum of products**, we combine the 1's to obtain:
  - **$F = x'z + xz'$**
- For the **product of sums**, we combine the 0's to obtain:
  - **$F' = xz + x'z'$**
- Taking the complement of *F*, we obtain the simplified function in product-of-sums form:
  - **$F = (x' + z')(x + z)$**



Entering product of sums form to map

- To enter a function expressed in product-of-sums form into the map, **use the complement** of the function to find the squares that are to be marked by 0's.
- *For example:*
  - $F = (A' + B' + C')(B + D)$
- First taking its complement:
  - $F' = ABC + B'D'$
- Then marking 0's in the squares representing the minterms of ***F'***. The remaining squares are marked with 1's.

3.5 Don't Care Conditions

- Real circuits don't always need to have an output defined for every possible input. (In practice, in some applications the function is not specified for certain combinations of the variables.)
  - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display 2^7 -1 patterns, but only ten of them are useful.
  - Functions that have unspecified outputs for some input combinations are called ***incompletely specified functions***
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a ***don't care*** condition.
- They are very helpful to us in K-map circuit simplification and don't-care conditions can be used on a map to **provide further simplification** of the Boolean expression.

3.5 Don't Care Conditions

- The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1.
- The function is equal to 0 for the rest of the minterms.
- This pair of conditions **assumes that** all the combinations of the values for the variables of the function are valid.
- **In practice, in some applications the function is not specified for certain combinations of the variables.**
  - **Ex:** The **four-bit binary code** for the **decimal digits** has **six combinations** that are not used and consequently are considered to be unspecified.
  - Functions that have unspecified outputs for some input combinations are called **incompletely specified functions**
    - ***Which creates unspecified minterms***
- To distinguish the **don't-care condition** from 1's and 0's, an **X** is used

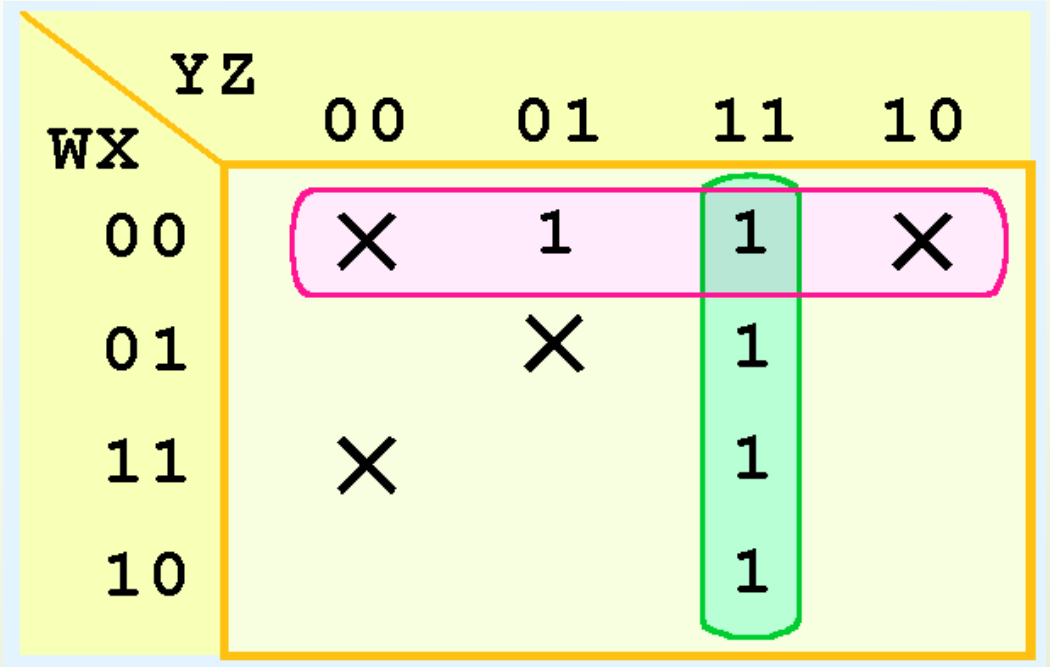
3.5 Don't Care Conditions

- In a Kmap, a don't care condition is identified by an *X* in the cell of the minterm(s) for the don't care inputs, as shown below.
- In performing the simplification, **we are free to include or ignore the *X*'s** when creating our groups.

WX \ YZ	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

3.5 Don't Care Conditions

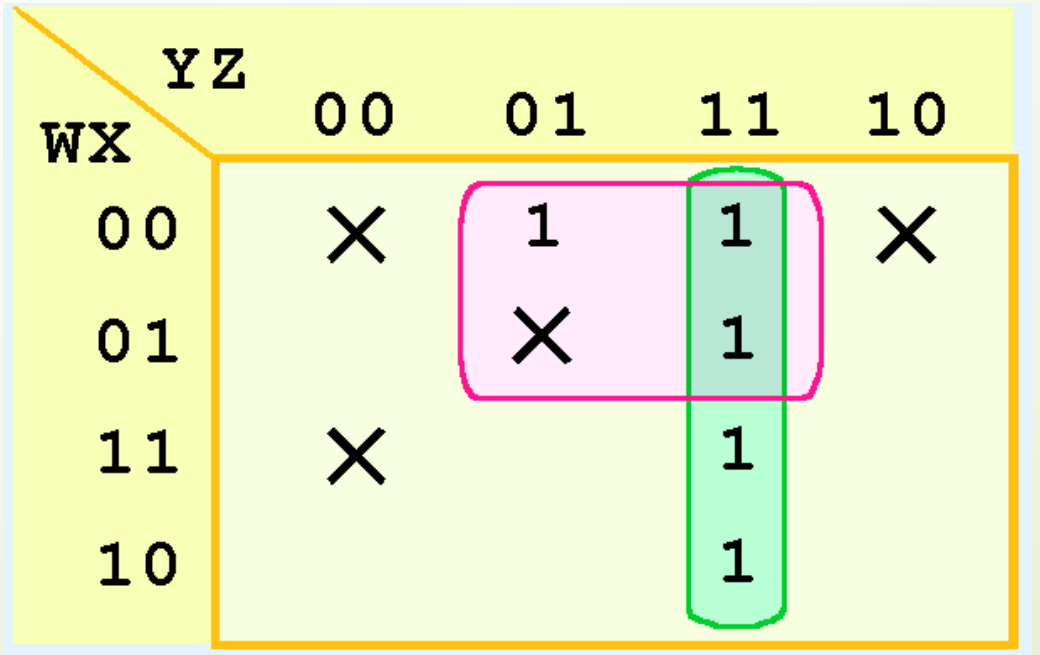
- In one grouping in the Kmap below, we have the function: $F(W,X,Y,Z) = \overline{W}\,\overline{Y} + YZ$



3.5 Don't Care Conditions

- A different grouping gives us the function:

$$F(W,X,Y,Z) = \bar{W}Z + YZ$$

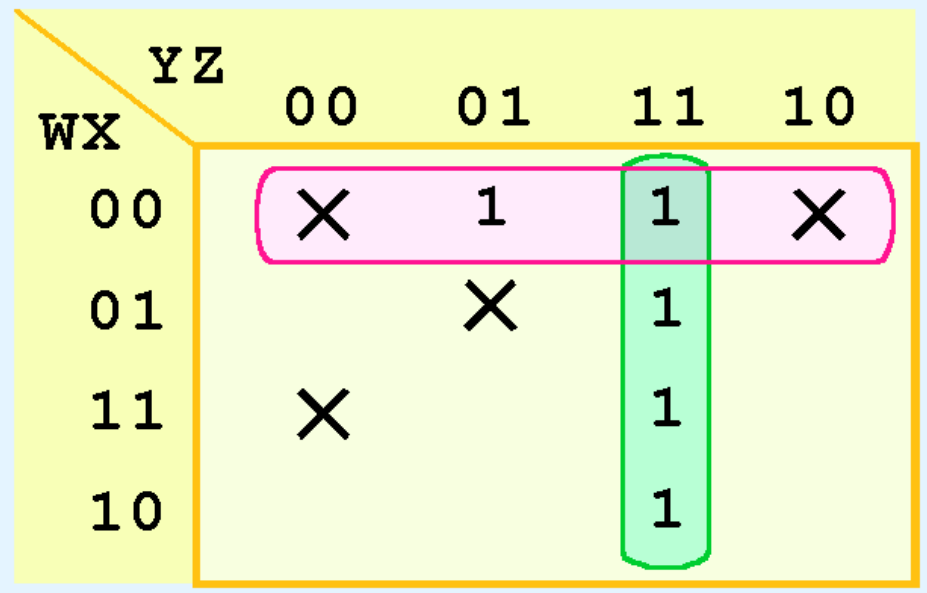


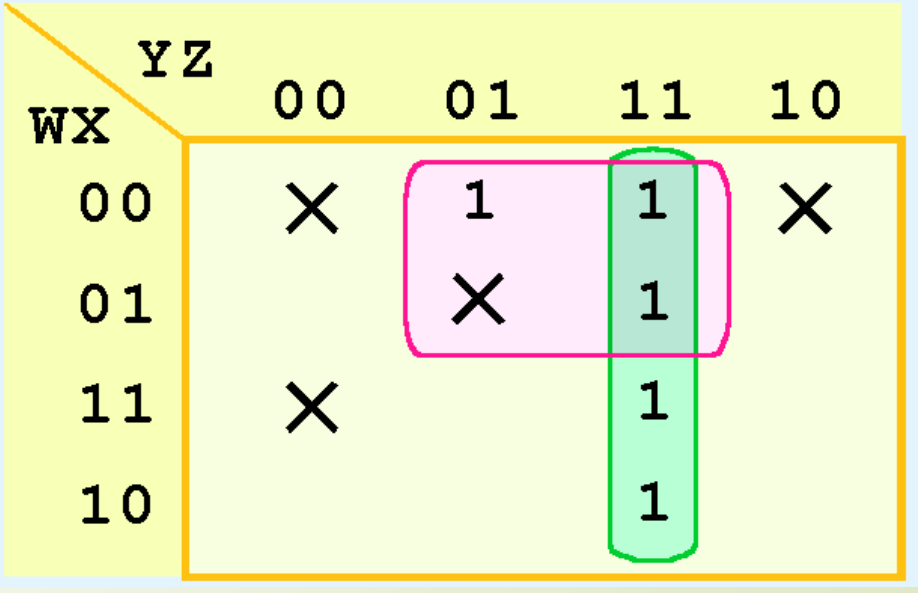
3.5 Don't Care Conditions

- The truth table of: $F(W,X,Y,Z) = \overline{W}\,\overline{Y} + YZ$

  differs from the truth table of: $F(W,X,Y,Z) = \overline{W}Z + YZ$

- However, the values for which they differ, are the inputs for which we have don't care conditions.



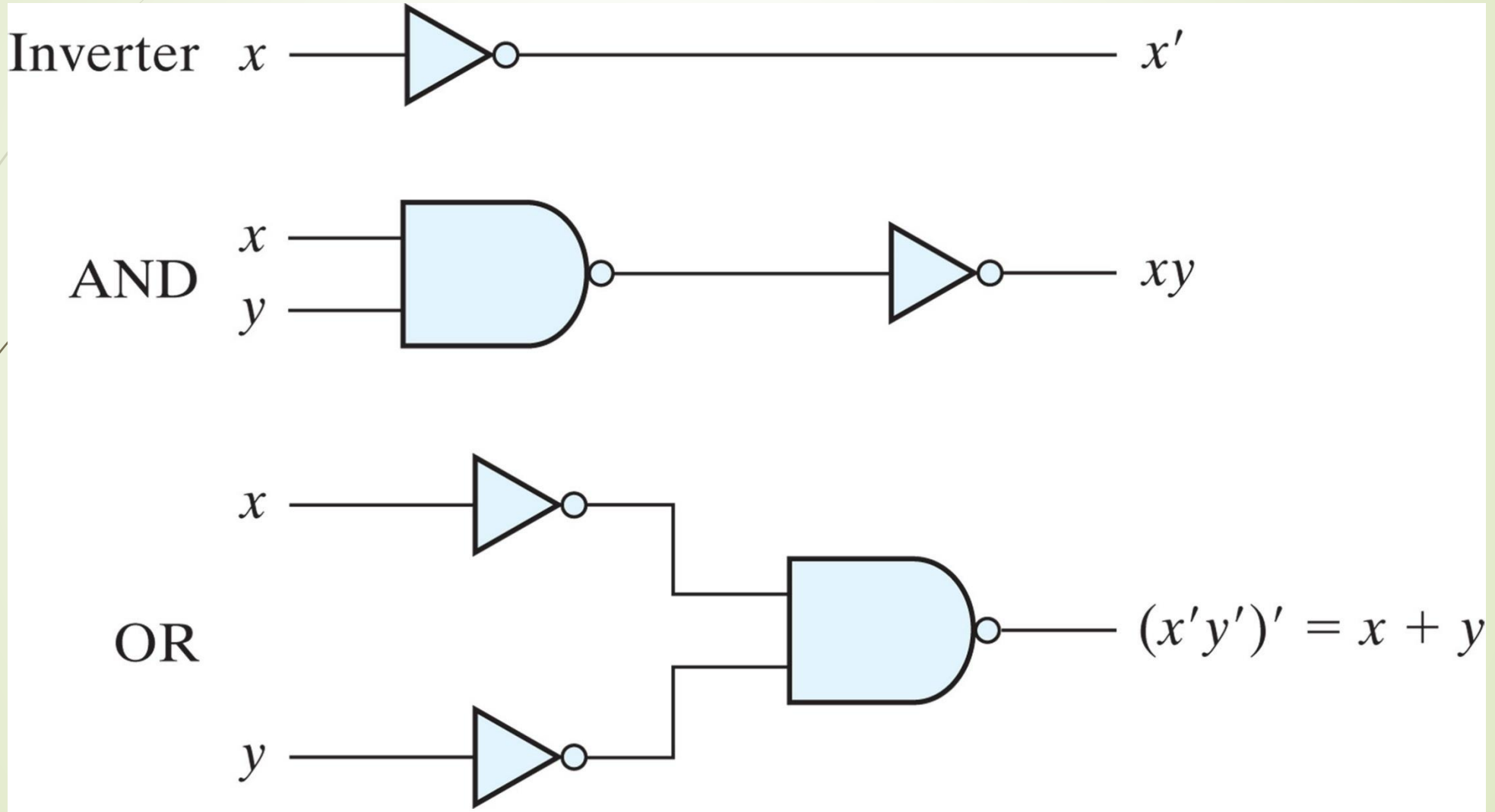


3.6 NAND and NOR Implementation

- Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates.
- NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families.

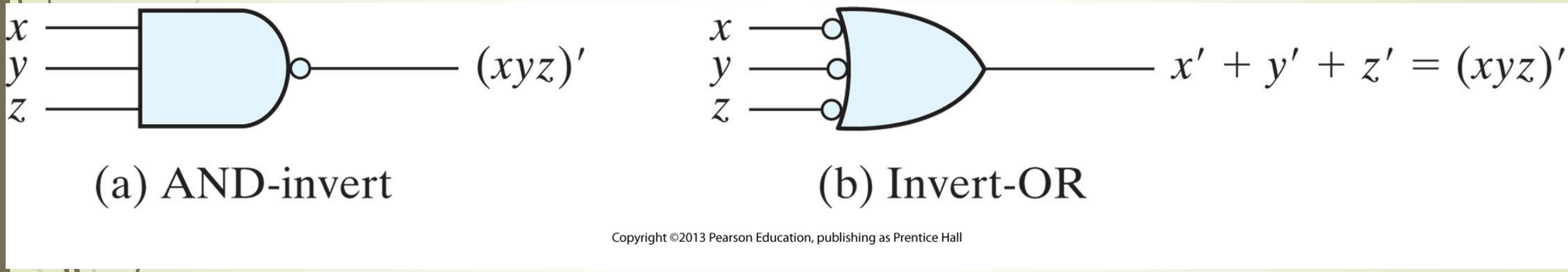
- Two-level forms
- Multilevel forms

Logic operations with NAND gates

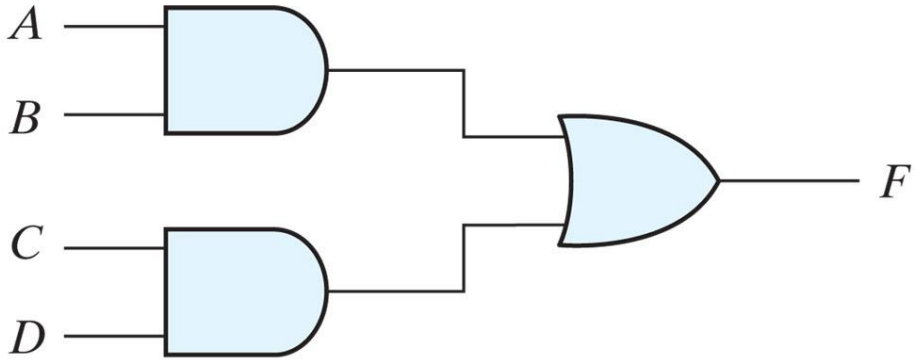


Logic operations with NAND gates

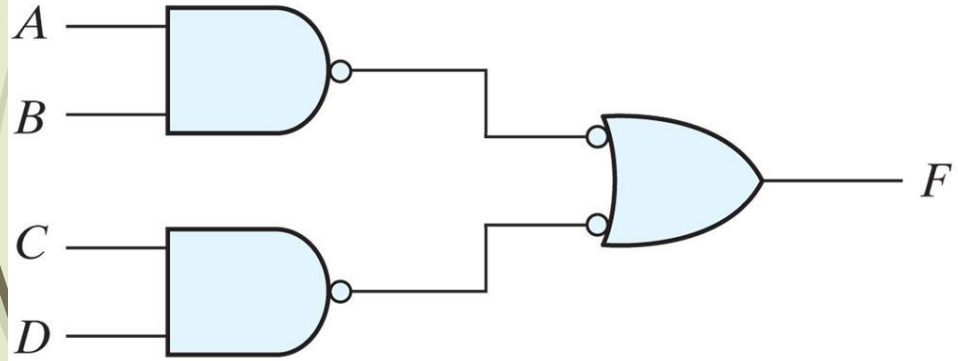
- Two graphic symbols for a three-input NAND gate



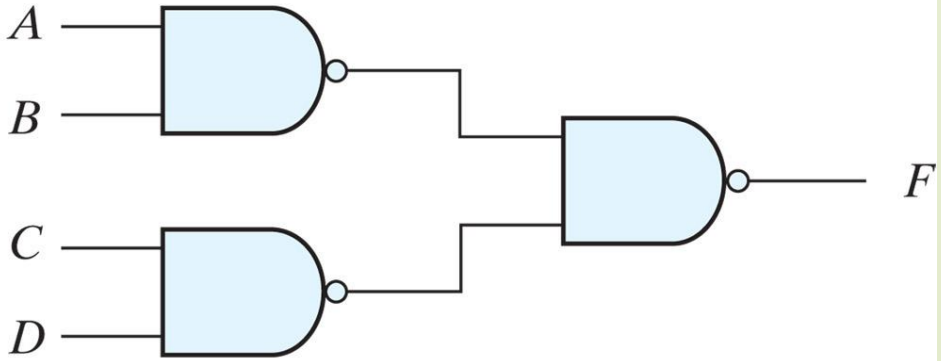
Two-level forms



(a)



(b)



(c)

The procedure for obtaining the logic diagram from a Boolean function

- Simplify the function and express it in sum-of-products form.
- Draw a NAND gate for each product term of the expression that has **at least two literals**. **The inputs to each NAND gate are the literals of the term**. This procedure produces a group of **first-level gates**.
- Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
- A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate.

NAND gate example

EXAMPLE 3.9

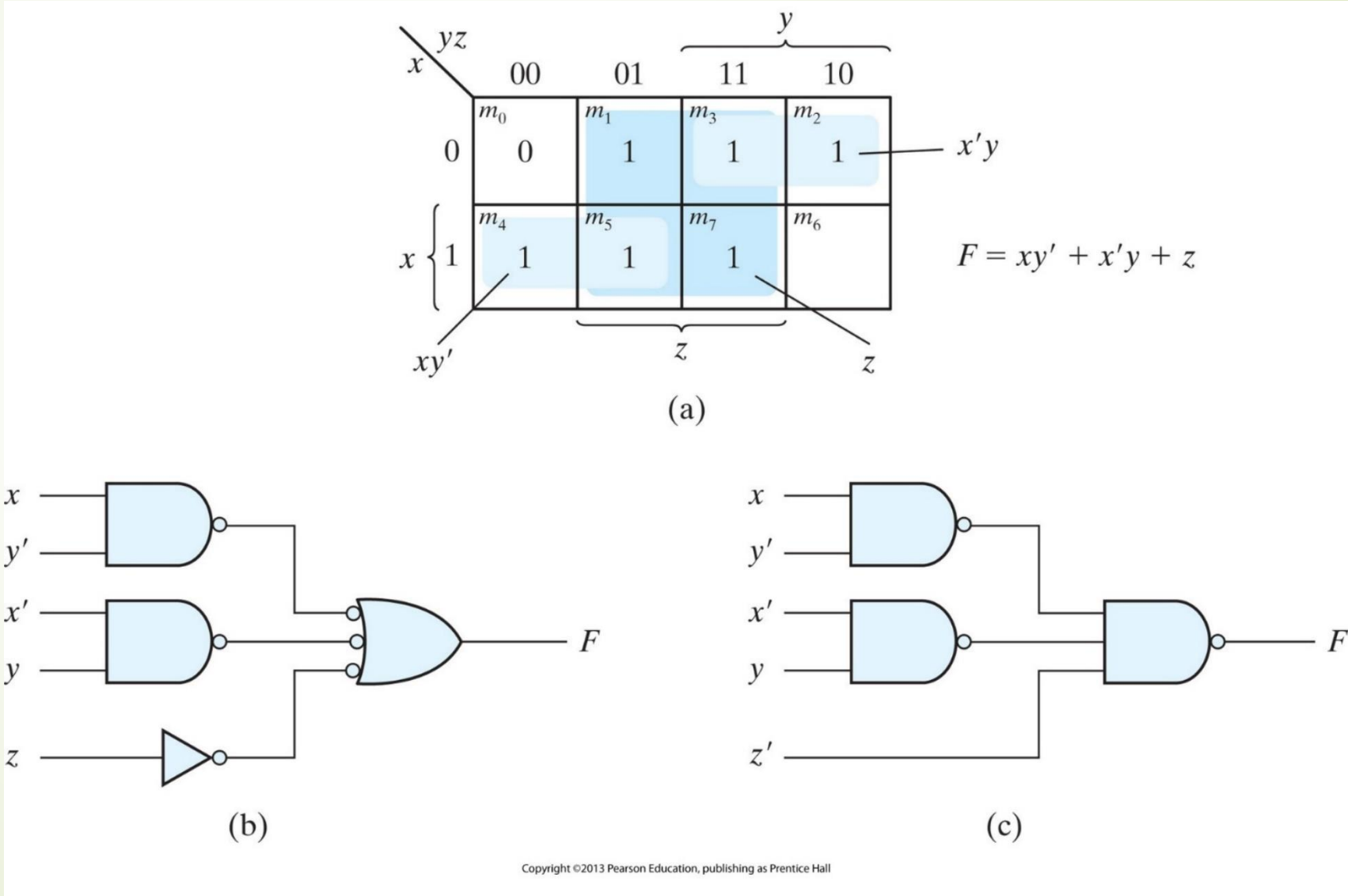
Implement the following Boolean function with NAND gates:

$$F(x, y, z) = (1, 2, 3, 4, 5, 7)$$

- The first step is to simplify the function into **sum-of-products** form.
  - F = xy' + x'y + z

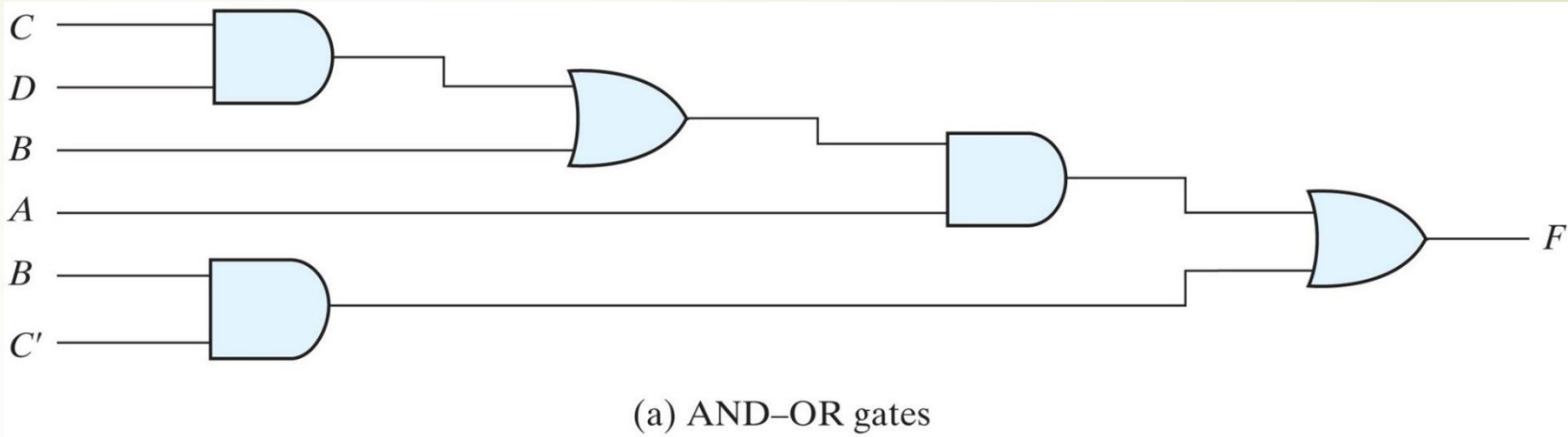
NAND gate example (3.9)

- F = xy' + x'y + z

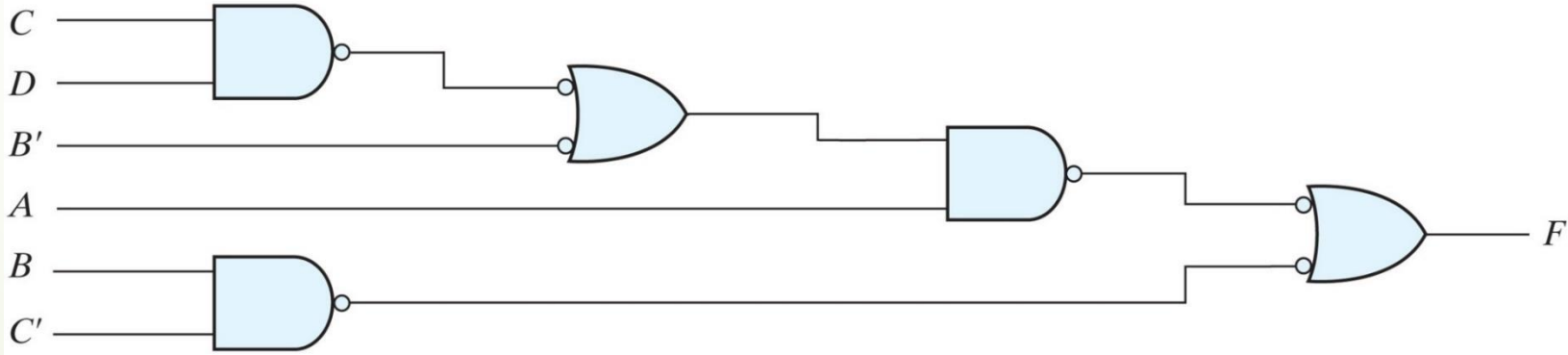


Multi-level forms

- $F = A(CD + B) + BC'$
- The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol.
- **Two bubbles on the same line**



(a) AND–OR gates



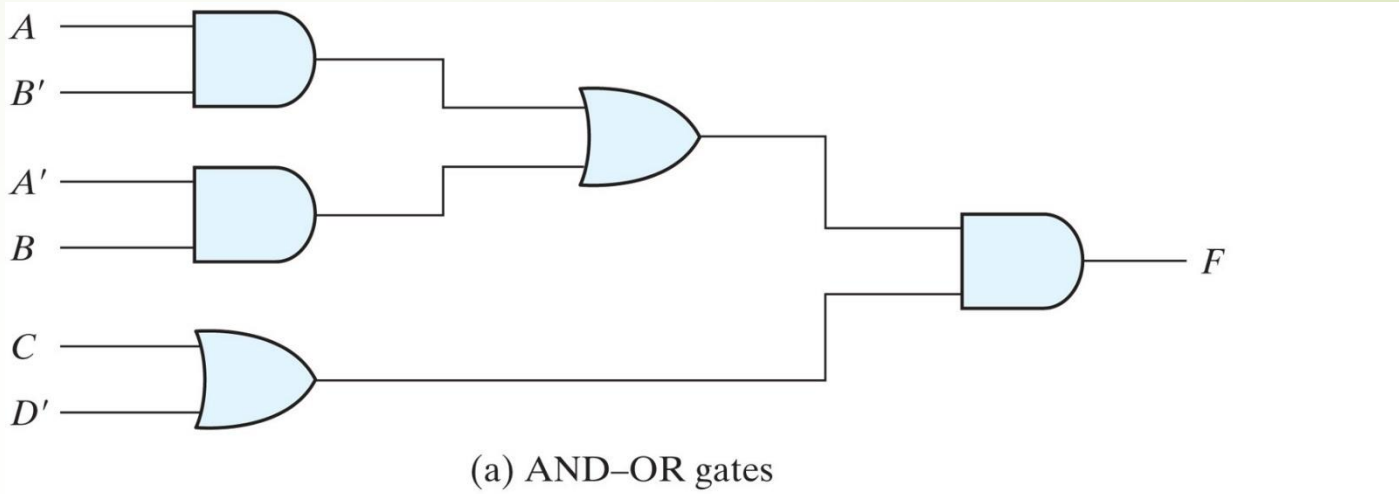
(b) NAND gates

The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation

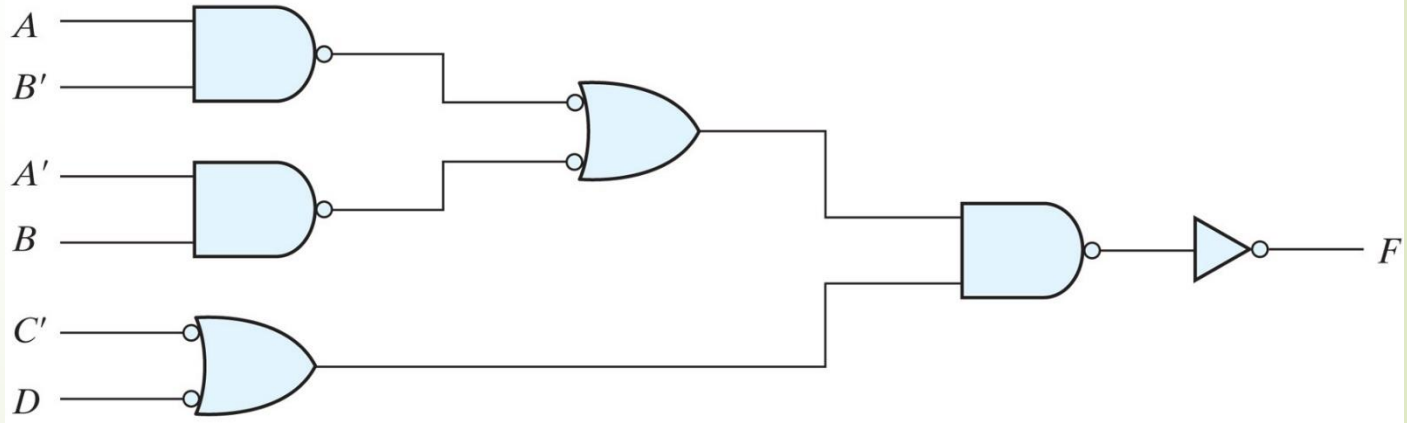
- Convert all AND gates to NAND gates with AND-invert graphic symbols
- Convert all OR gates to NAND gates with invert-OR graphic symbols
- Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

Converting AND-OR diagram to NAND

- $F = (AB' + A'B)(C + D')$



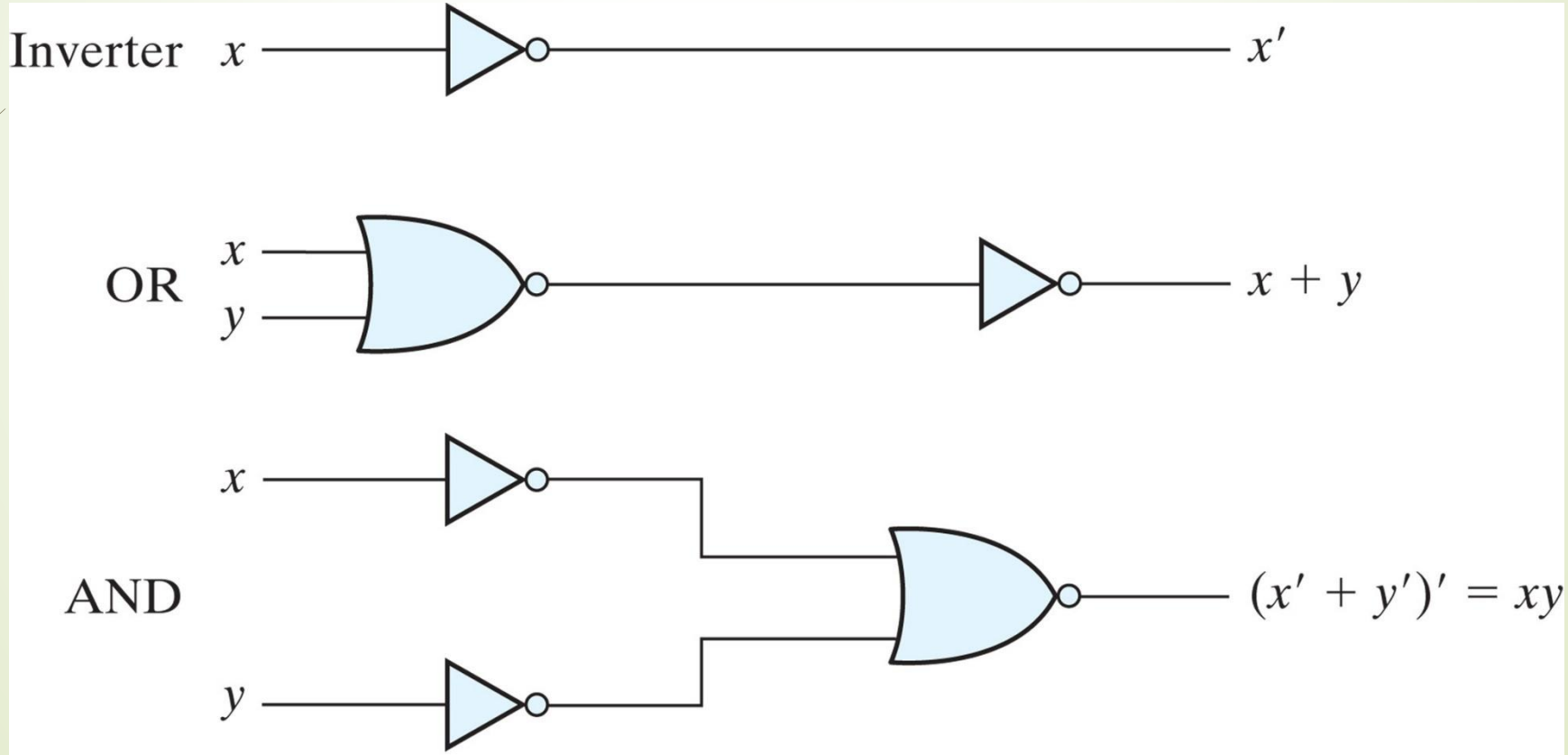
(a) AND–OR gates



(b) NAND gates

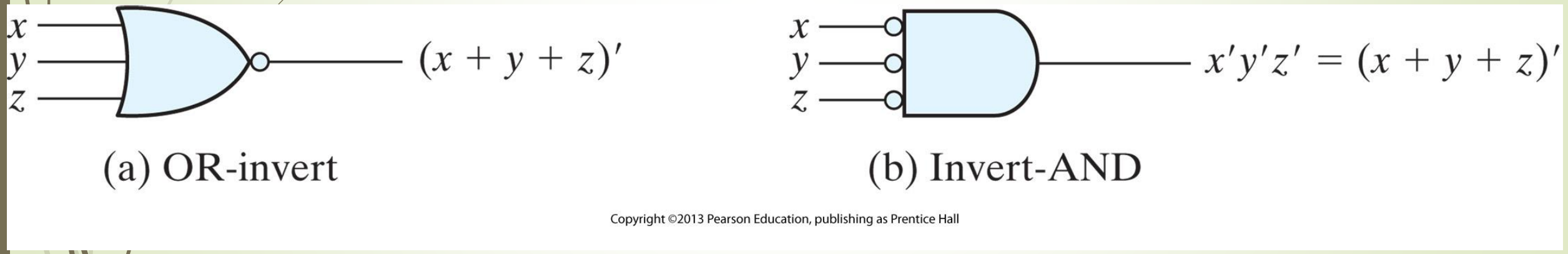
Logic operations with NOR gates

- The NOR operation is the dual of the NAND operation



Logic operations with NOR gates

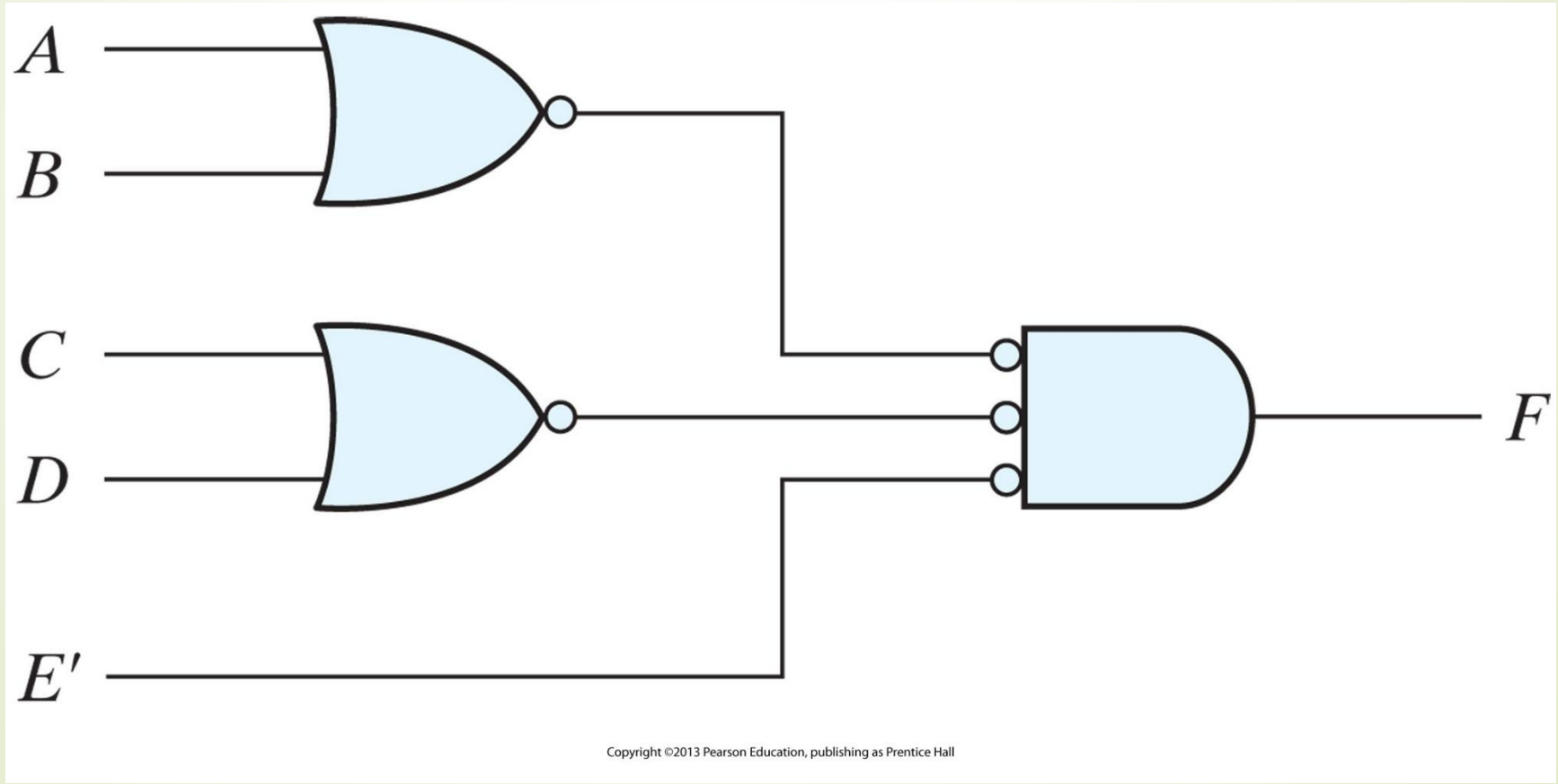
- Two graphic symbols for the NOR gate



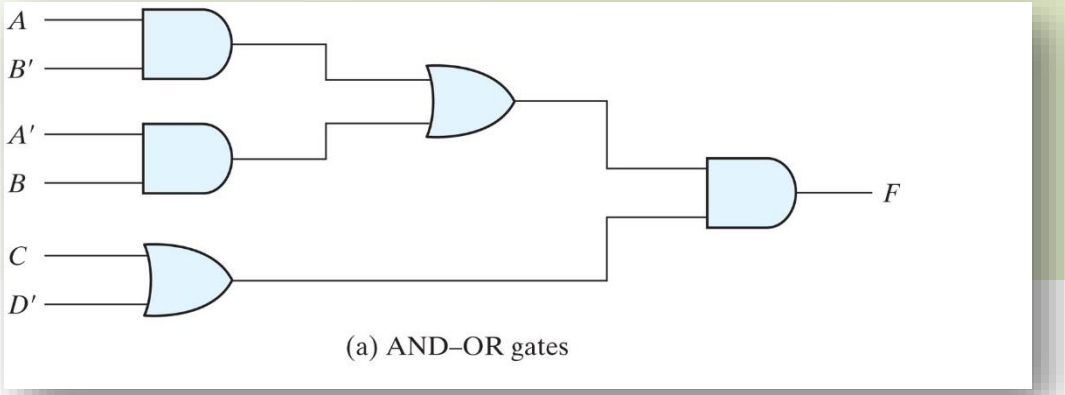
(a) OR-invert (b) Invert-AND

Example NOR implementation

- $F = (A + B)(C + D)E$

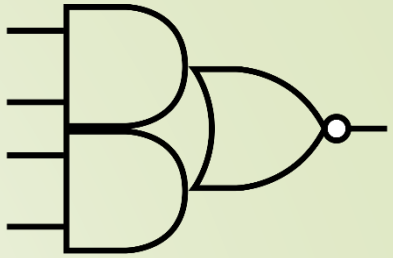


Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

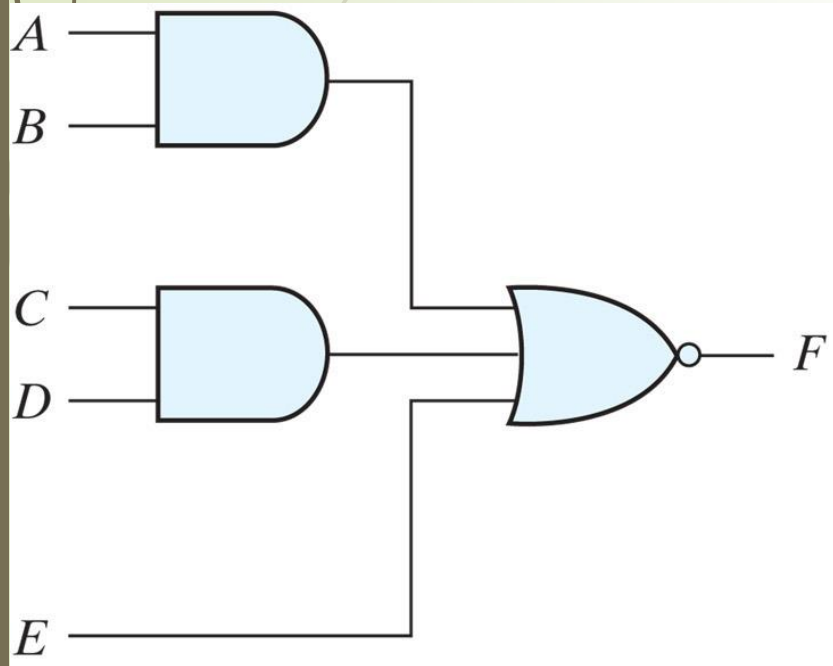


(a) AND–OR gates

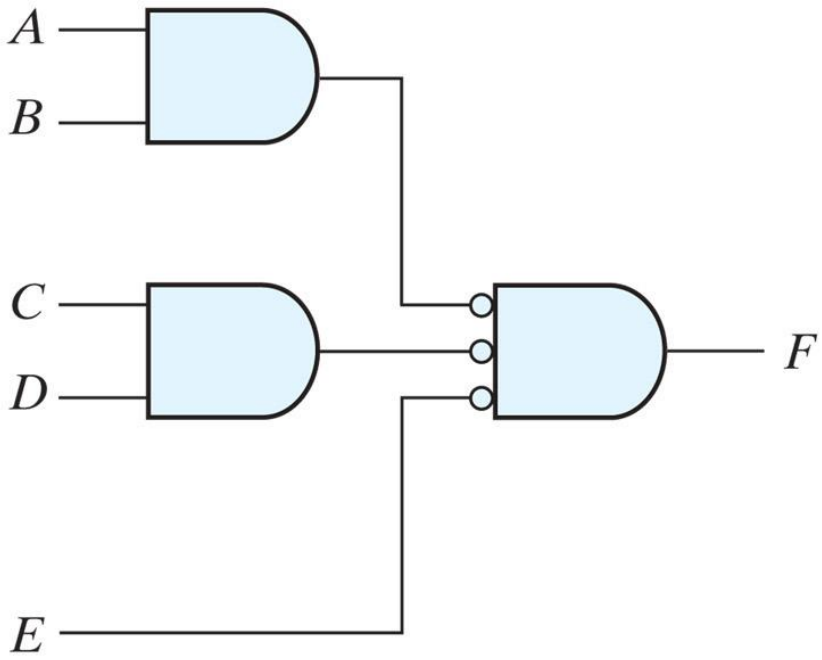
AND–OR–INVERT Implementation



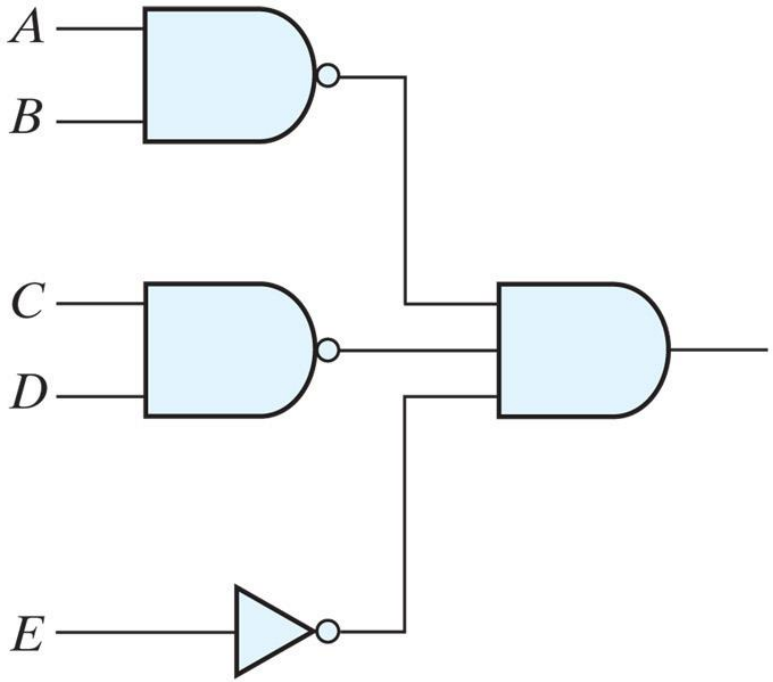
- $F = (AB + CD + E)'$



(a) AND–NOR



(b) AND–NOR



(c) NAND–AND

Odd Function

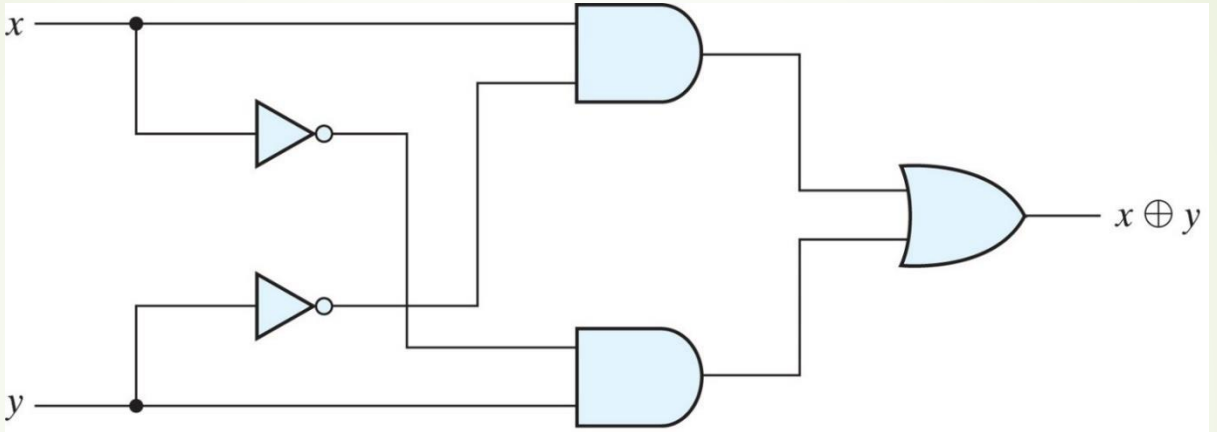
- Three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1.
- Contrary, to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1.
- Therefore, multiple-variable exclusive-OR operation is defined as an ***odd function***

INPUTS		OUTPUTS
A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

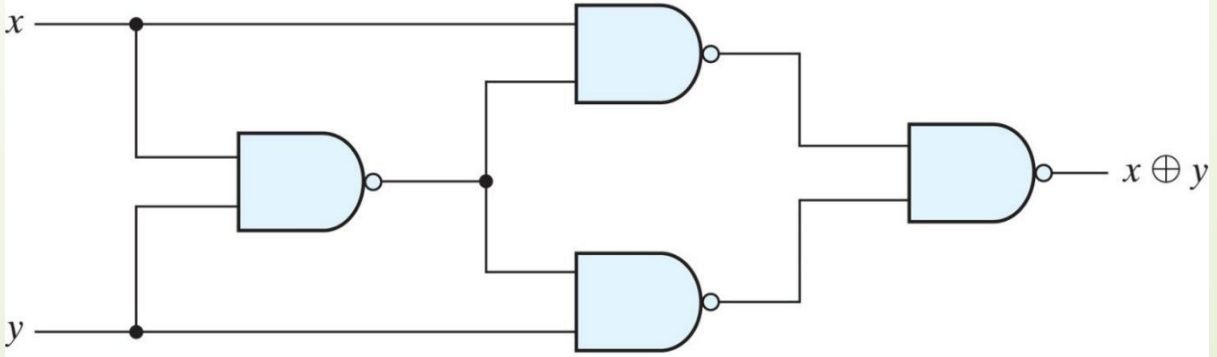
3-input XOR gate

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Exclusive OR implementations



(a) Exclusive-OR with AND–OR–NOT gates



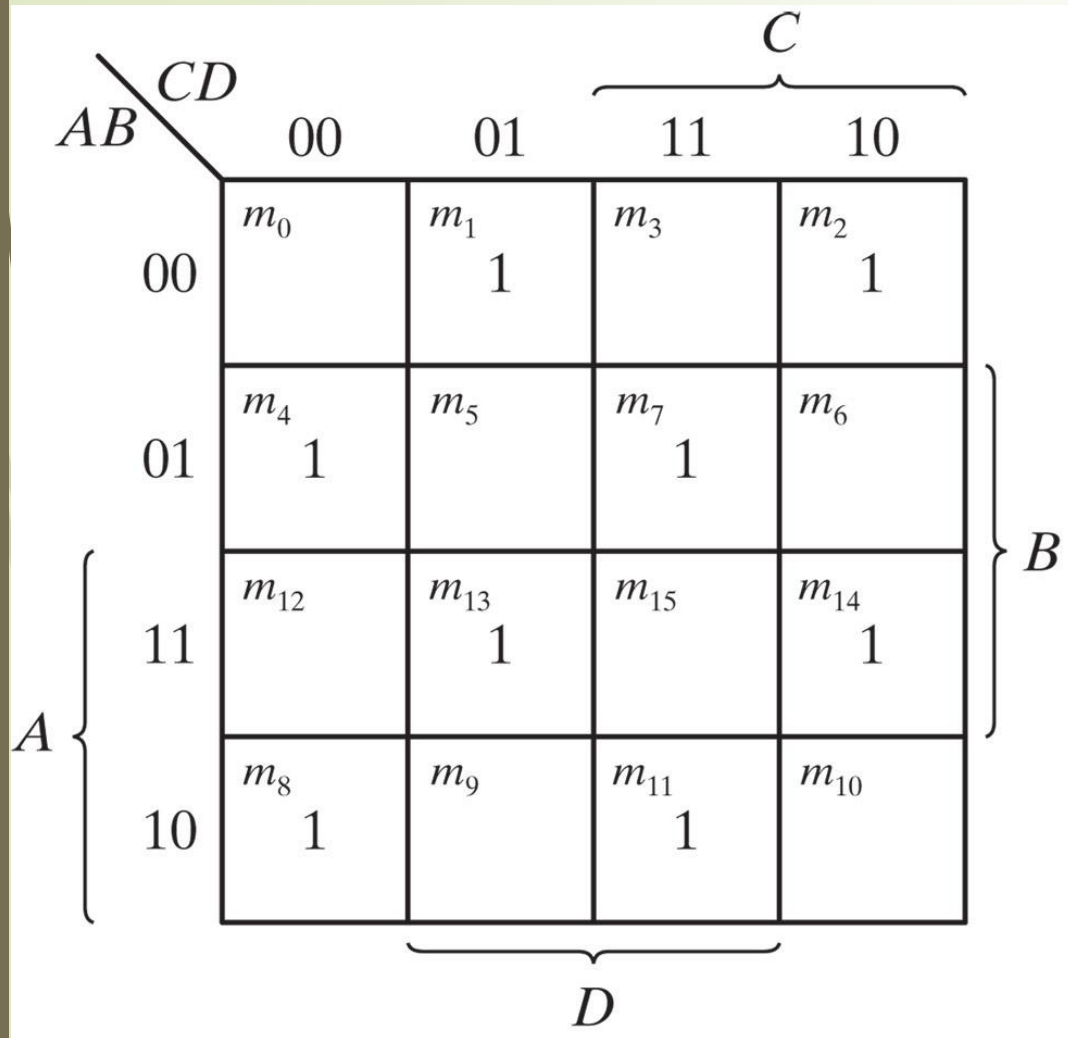
(b) Exclusive-OR with NAND gates

Odd and Even functions using XOR

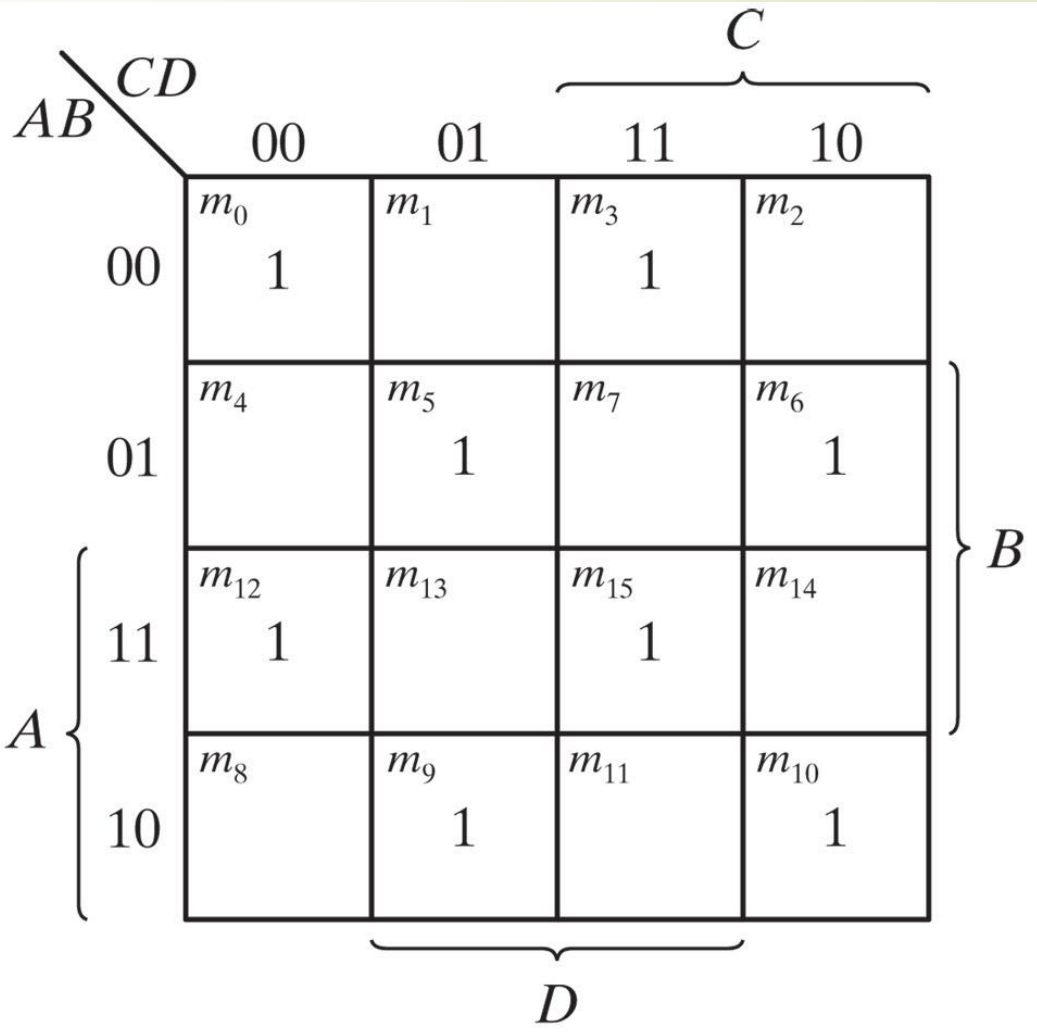
- Consider now the four-variable exclusive-OR operation

$$
\begin{aligned}
A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\
&= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\
&= \Sigma(1, 2, 4, 7, 8, 11, 13, 14)
\end{aligned}
$$

Odd and Even functions using XOR



(a) Odd function $F = A \oplus B \oplus C \oplus D$



(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

Parity Generation and Checking

- Exclusive-OR functions are very useful in systems requiring error detection and correction codes
- **A parity bit** is used for the **purpose of detecting errors** during the transmission of binary information
- A parity bit is an extra bit included with a binary message to make the number of 1's either **odd** or **even**

Parity Generation and Checking

- The message, including the parity bit, is transmitted and then **checked at the receiving end for errors.**
- **Parity Generator**
  - The circuit that generates the parity bit in the transmitter is called a **parity generator.**
- **Parity Checker**
  - The circuit that checks the parity in the receiver is called a **parity checker.**

Even-Parity-Generator Truth Table

- *P* can be expressed as a three-variable exclusive-OR function

$$P = x \oplus y \oplus z$$

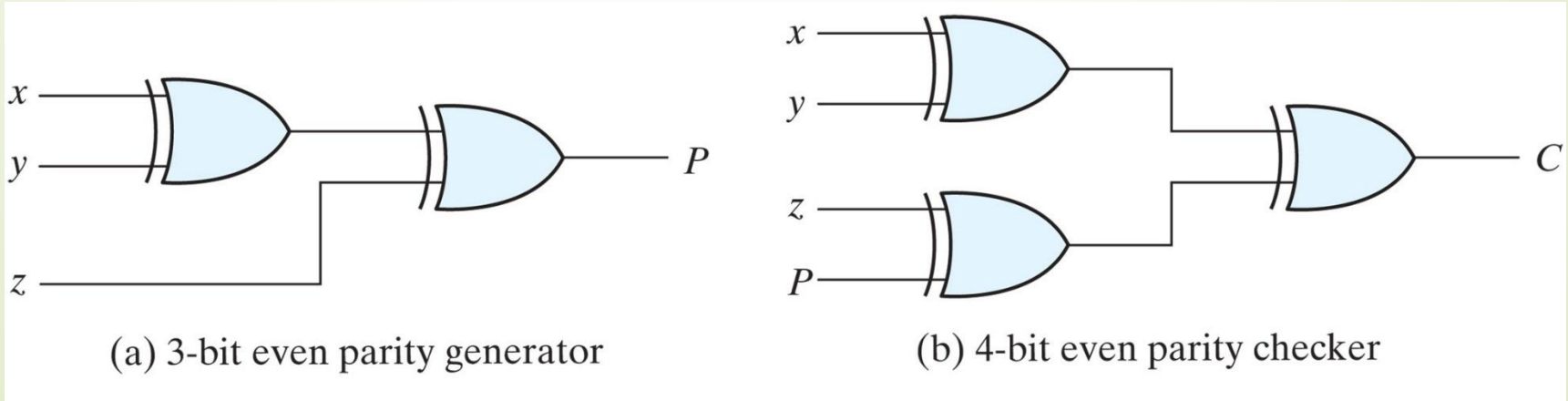
Table 3.3
Even-Parity-Generator Truth Table

Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Hamming Error Correction Codes

Logic diagram of a parity generator and checker

- Since the information was transmitted with **even parity**, the **four bits received** must have an **even number of 1's.**
- An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission
- The output of the parity checker, denoted by C , will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1's



(a) 3-bit even parity generator

(b) 4-bit even parity checker

Even-Parity-Checker Truth Table

- The parity checker can be implemented with exclusive-OR gates

$$C = x \oplus y \oplus z \oplus P$$

Table 3.4
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Hardware Description Language (HDL)

- Manual methods for designing logic circuits are feasible only when the circuit is **small**
- Prototype integrated circuits are too **expensive** and **time consuming** to build
- A **hardware description language** (**HDL**) is a computer-based language that describes the hardware of digital systems in a textual form
- ***Design entry*** creates an HDL-based description of the functionality that is to be implemented in hardware

Hardware Description Language (HDL)

- ***Logic simulation*** displays the behavior of a digital system through the use of a computer.
  - Detects functional errors in a design
  - Testing the functionality of the design is called a ***test bench***
    - ***Have i tried all the cases?***
    - ***Have i exercised every path?***
    - ***is every option tested?***
- Simulation checks two properties
  - Functional correctness
    - is the logic correct
  - Timing correctness
    - is the logic interconnection times are correct

Hardware Description Language (HDL)

- A simulator **interprets** the HDL description and **produces readable output**, such as a **time-ordered sequence of input and output signal values**.
- ***Logic synthesis (compiling)*** is the process of deriving a list of physical components and their interconnections (called a ***netlist***) from the model of a digital system described in an HDL.
  - Creates a database describing the elements and structure of a circuit
  - The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL.

Hardware Description Language (HDL)

- ***Timing verification*** confirms that the fabricated, integrated circuit will operate at a specified speed.
  - Because each logic gate in a circuit has a **propagation delay**, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit.
  - Timing verification checks each signal path to verify that it is not compromised by propagation delay
- In VLSI circuit design, ***fault simulation*** compares the behavior of an **ideal circuit** with the behavior of a circuit that contains a process-induced flaw
  - Fault simulation is used to identify input stimuli that can be used to reveal the difference between the **faulty circuit** and the **fault-free circuit**
  - test patterns will be used to test fabricated devices to ensure that only good devices are shipped to the customer

HDL Types

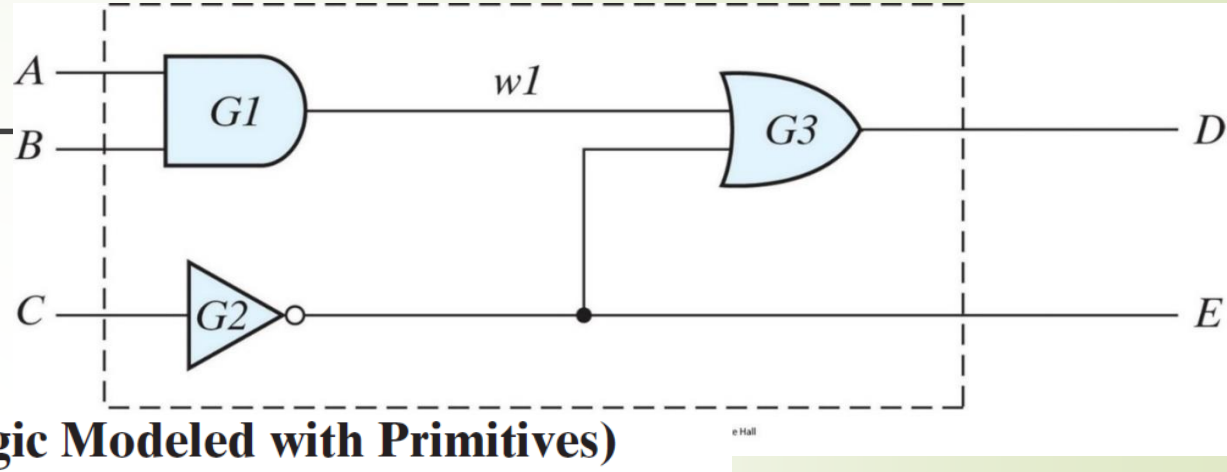
- There are two standard HDLs that are supported by the IEEE
  - **VHDL**
    - V stands for VHSIC (Very High-Speed Integrated Circuit)
    - VHDL is more difficult to learn than Verilog.
  - **Verilog**
    - The Verilog HDL was initially approved as a standard HDL in 1995 (revised in 2001,2005)

HDL Module Declaration

- 100 **keywords**
- **Keywords** are predefined lowercase identifiers
  - **E.g.: module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**
- Comments
  - // used for comments
  - /* */ for multiline comments
- Verilog is case sensitive
  - **not** is not the same as NOT
- Identifiers must start with an alphabetic character or an underscore, but they cannot start with a number
- The term ***module*** refers to the text enclosed by the keyword pair **module** . . . **endmodule**
- It is declared by the keyword **module** and must always be **terminated** by the keyword endmodule

HDL Module Declaration

- Circuit to demonstrate an HDL

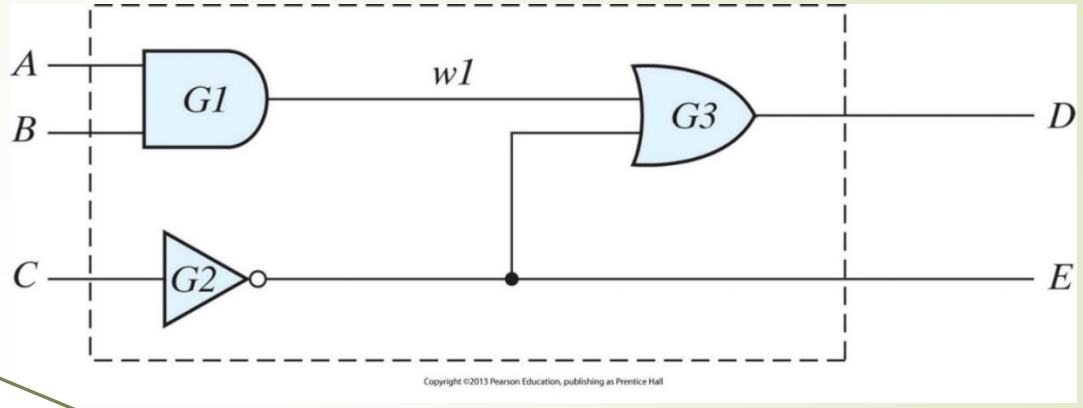


HDL Example 3.1 (Combinational Logic Modeled with Primitives)

```
// Verilog model of circuit of Figure 3.35. IEEE 1364–1995 Syntax

module  Simple_Circuit (A, B, C, D, E);
  output        D, E;
  input         A, B, C;
  wire          w1;

  and           G1 (w1, A, B); // Optional gate instance name
  not           G2 (E, C);
  or            G3 (D, w1, E);
endmodule
```

HDL Module Declaration

- The ***port list*** of a module is the interface between the module and its environment.
  - In this example, the ports are the **inputs** and **outputs** of the circuit.
  - The **port list** is enclosed in **parentheses**, and **commas** are used to separate elements of the list
  - The statement is terminated with a semicolon (;)
- Keywords **input** and **output** specify which of the ports are inputs and which are outputs
- Internal connections are declared as **wires**.
- ***Primitive* gates**, each identified by a descriptive keyword (**and, not, or**)



HDL Example 3.1 (Combinational Logic Modeled with Primitives)

```
// Verilog model of circuit of Figure 3.35. IEEE 1364–1995 Syntax

module  Simple_Circuit (A, B, C, D, E);
  output        D, E;
  input         A, B, C;
  wire          w1;

  and           G1 (w1, A, B); // Optional gate instance name
  not           G2 (E, C);
  or            G3 (D, w1, E);
endmodule
```

No semicolon

Each of the gate is called as **gate instance**

HDL Module Declaration

- **Predefined primitives are not declared**, because their definition is specified by the language and is not subject to change by the user.
- The **sequential ordering** of the statements instantiating gates in the model **has no significance** and **does not specify a sequence of computations**.
- Verilog model is a ***descriptive* model**

Gate Delays

- All physical circuits exhibit a **propagation delay** between the transition of an input and a resulting transition of an output.
- In Verilog, the **propagation delay** of a gate is specified in terms of ***time units*** and by the symbol **#**
- The numbers associated with **time delays**
- The association of a time unit with physical time is made with the **'timescale compiler directive**
  - Compiler directives start with the (') back quote, or grave accent symbol.
- E.g.:
  - '**timescale** 1ns/100ps
    - The first number specifies the unit of measurement for time delays
    - The second number specifies the **precision** for which the delays are rounded off, in this case to 0.1 ns.

Gate Delays

- The **and**, **or** , and **not** gates have a time delay of 30, 20, and 10 ns

HDL Example 3.2 (Gate-Level Model with Propagation Delays)

```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
  output D, E;
  input  A, B, C;
  wire   w1;

  and          #(30) G1 (w1, A, B);
  not          #(10) G2 (E, C);
  or           #(20) G3 (D, w1, E);
endmodule
```

Table 3.5 Output of Gates after Delay

- If the circuit is simulated and the inputs change from *A* , *B* , *C* = 0 to *A* , *B* , *C* = 1
  - and #(30)
  - not #(10)
  - or #(20)

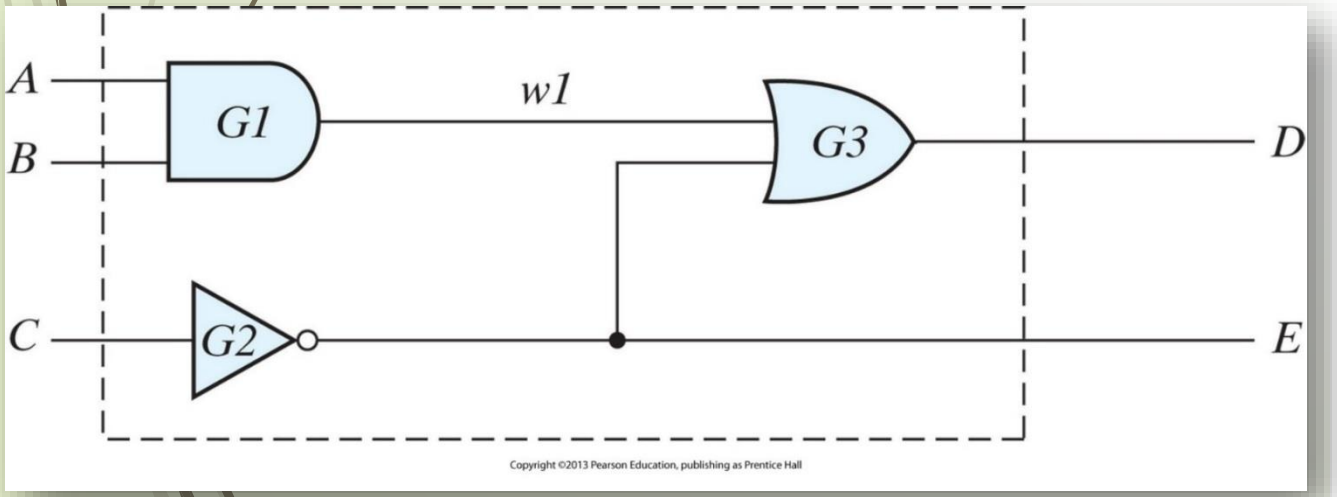
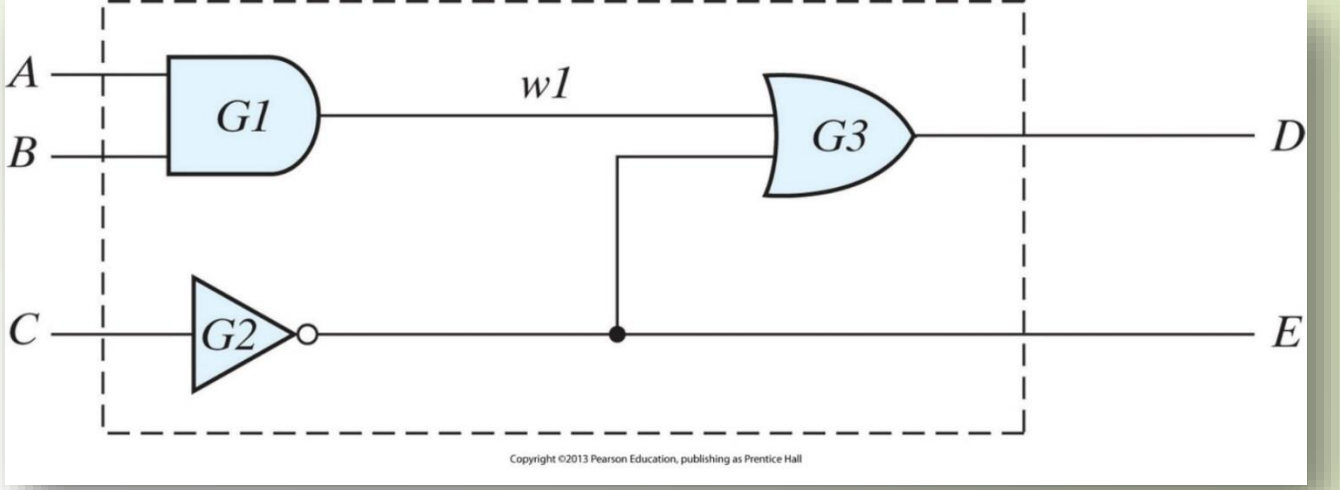


Table 3.5
Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	0 0 0	1 0 1
Change	—	1 1 1	1 0 1
	10	1 1 1	0 0 1
	20	1 1 1	0 0 1
	30	1 1 1	0 1 0
	40	1 1 1	0 1 0
	50	1 1 1	0 1 1

Test bench

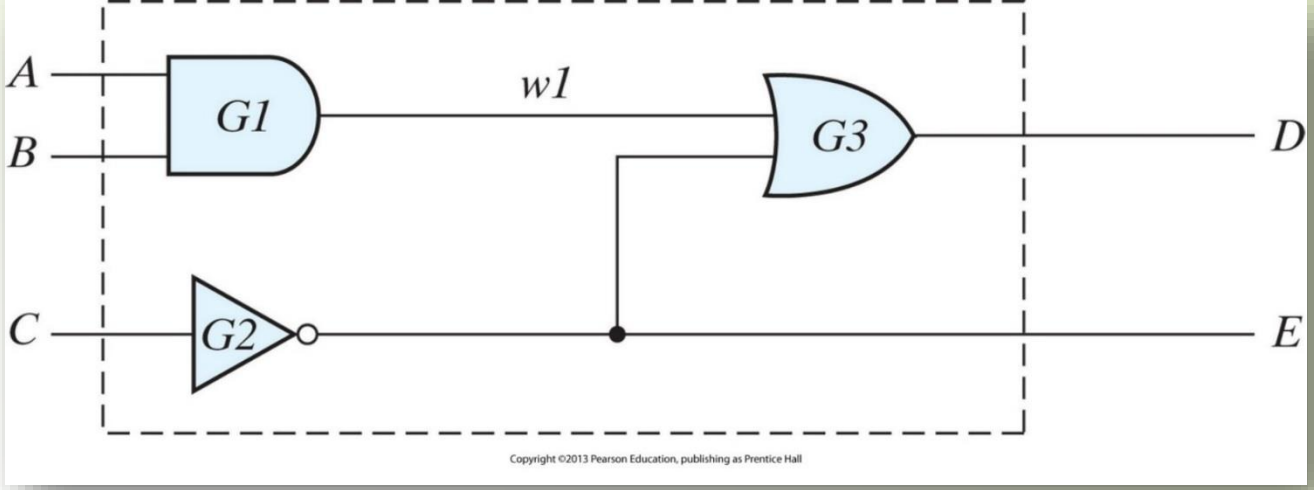
- Test benchs has no input or output ports
- No interaction with the environment
- the **inputs** to the circuit are declared with keyword **reg**
- the **outputs** are declared with the keyword **wire**
- The module *Simple_Circuit_prop_delay* is instantiated with the unique instance name **M1**




```
// Test bench for Simple_Circuit_prop_delay

module  t_Simple_Circuit_prop_delay;
  wire     D, E;
  reg      A, B, C;

Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

  initial #200 $finish;
endmodule
```

Test bench

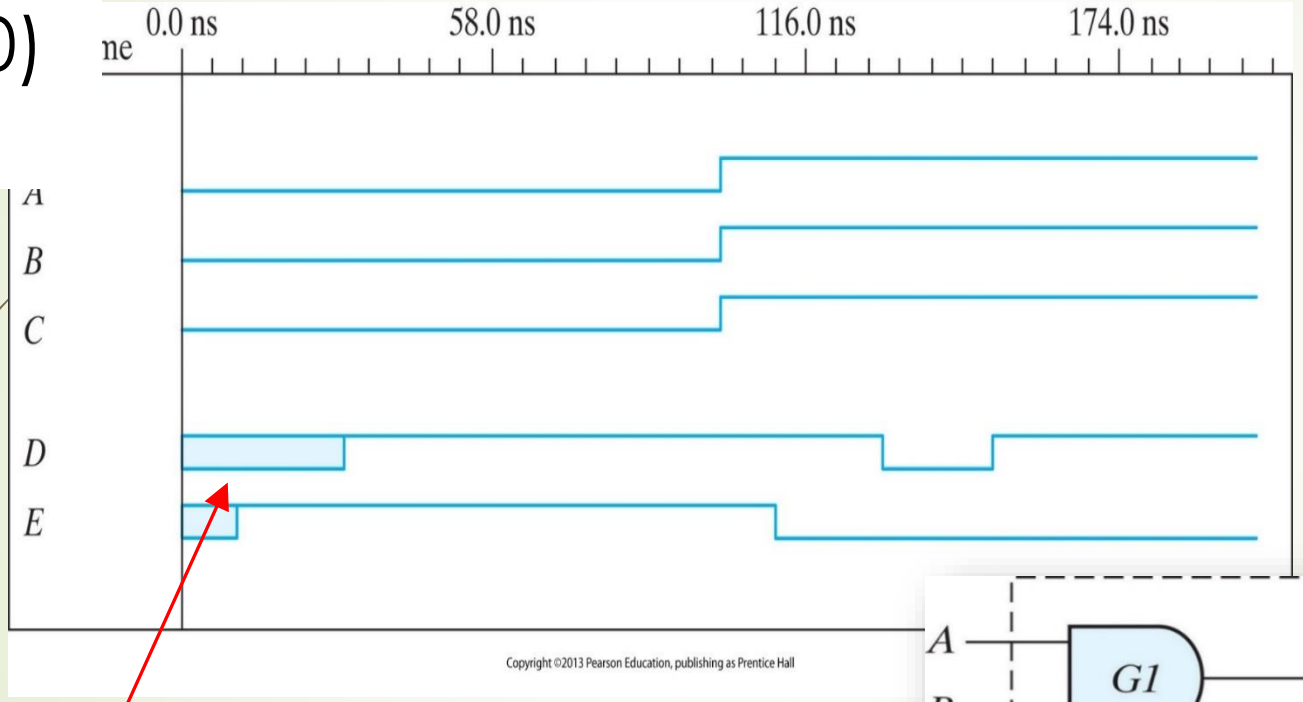
- The **initial** keyword is used with a set of statements that begin executing when the simulation is initialized
- Statements are executed in sequence, **left to right**, from **top to bottom**
- **1'b0** which signifies one binary digit with a value of **0**
- **After** 100 ns, the inputs change to *A*, *B*, *C* = 1.
- After another 100 ns, the simulation terminates at time 200 ns

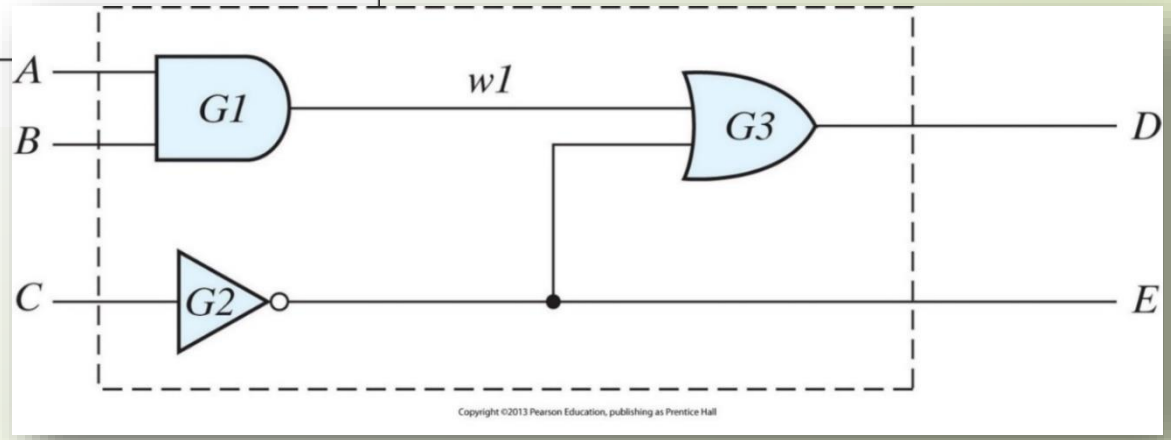



```
// Test bench for Simple_Circuit_prop_delay

module  t_Simple_Circuit_prop_delay;
  wire    D, E;
  reg     A, B, C;

Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

  initial #200 $finish;
endmodule
```

FIGURE 3.36 Simulation output of HDL Example 3.3

and #(30)
not #(10)
or #(20)





Unknown state

Boolean Expressions

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

- Keyword **assign** followed by a Boolean Expression
- Verilog uses the symbols (**&&**), (**|**), and (**!**) for **AND**, **OR**, and **NOT** (complement)

HDL Example 3.4 (Combinational Logic Modeled with Boolean Equations)

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
  output      E, F;
  input       A, B, C, D;

  assign E = A || (B && C) || ((!B) && D);
  assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

User Defined Primitives

It is declared with the keyword **primitive** , followed by a name and port list

There can be only one output, and it **must be listed first in the port list** and declared with keyword **output**

HDL Example 3.5 (User-Defined Primitive)

```
// Verilog model: User-defined Primitive
primitive  UDP_02467 (D, A, B, C);
  output  D;
  input    A, B, C;
//Truth table for D 5 f (A, B, C) 5 Σ(0, 2, 4, 6, 7);
  table
//      A     B     C     :     D     // Column header comment
        0     0     0     :     1;
        0     0     1     :     0;
        0     1     0     :     1;
        0     1     1     :     0;
        1     0     0     :     1;
        1     0     1     :     0;
        1     1     0     :     1;
        1     1     1     :     1;
  endtable
endprimitive
```

There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table

The truth table is enclosed within the keywords **table** and **endtable**

The values of the inputs are listed in order, ending with a colon (:)

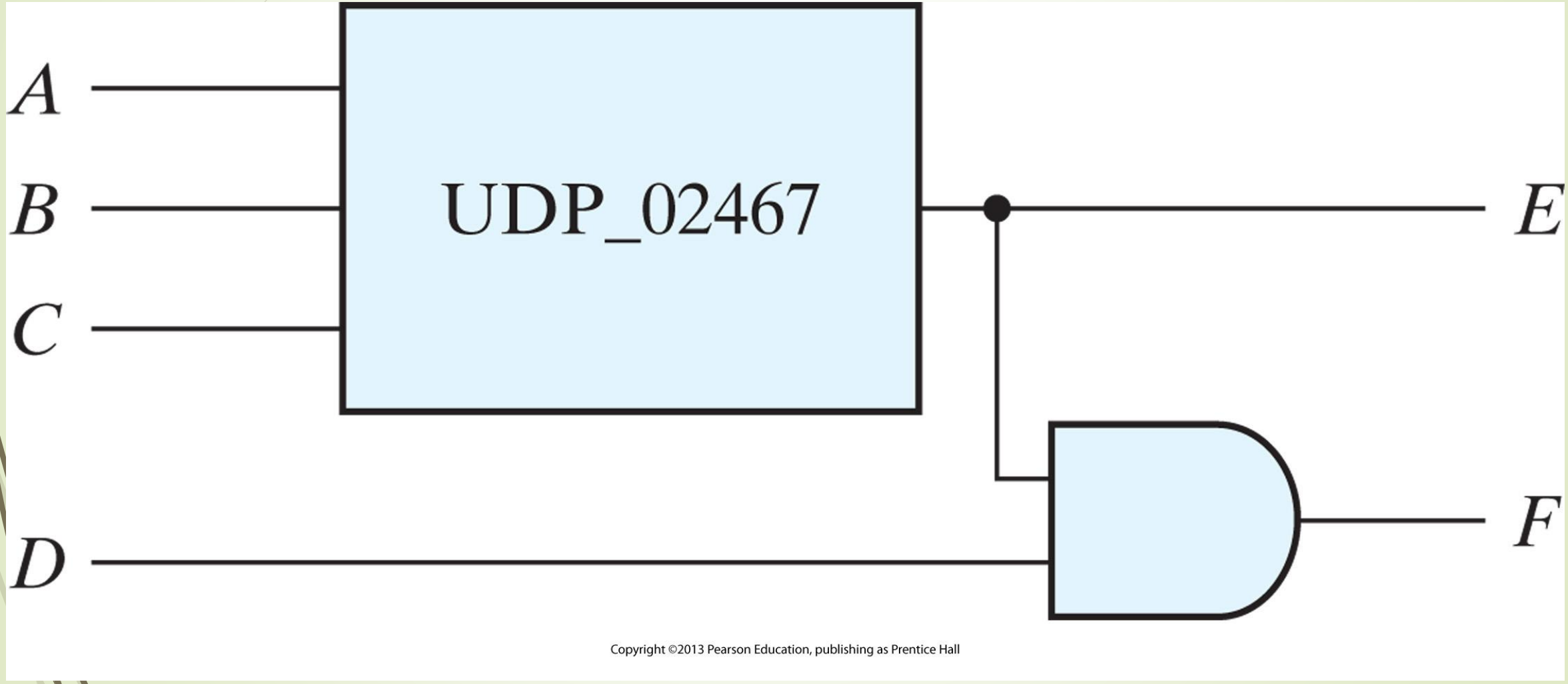
The output is always the last entry in a row and is followed by a semicolon (;)

The declaration of a UDP ends with the keyword **endprimitive**.

User Defined Primitives

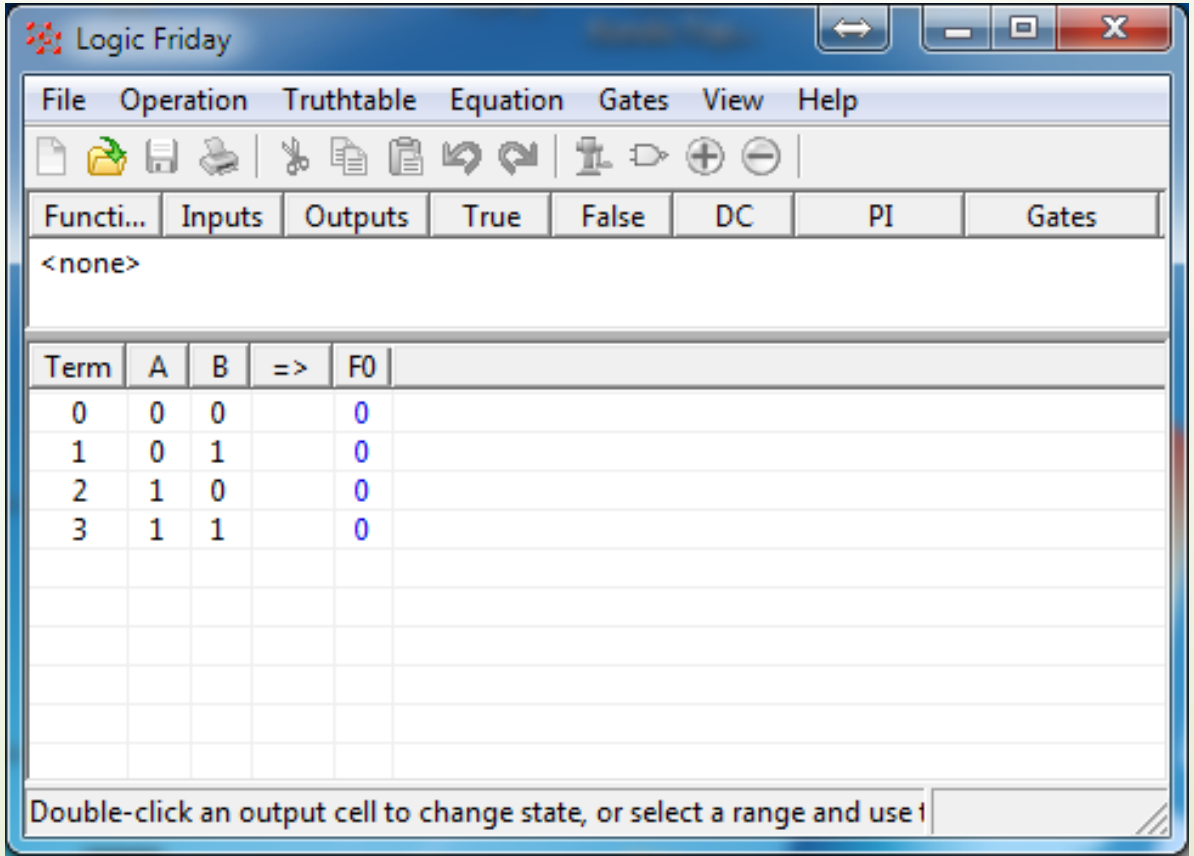
```
// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module Circuit_with_UDP_02467 (e, f, a, b, c, d);
  output        e, f;
  input         a, b, c, d

  UDP_02467             (e, a, b, c);
  and                   (f, e, d);          // Option gate instance name omitted
endmodule
```

User Defined Primitives



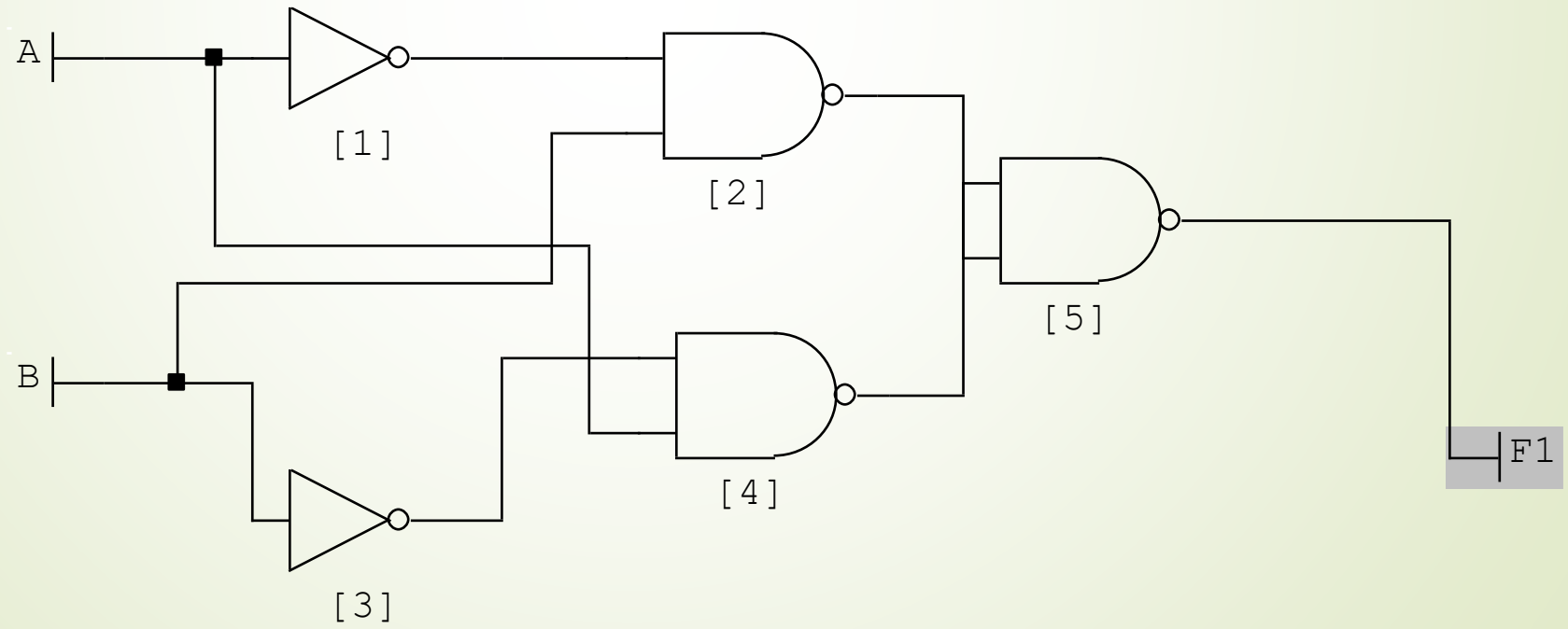
Logic Friday

- Logic Friday is a freeware tool for students, hobbyists, and engineers who work with legacy digital logic circuits based on standard IC packages.



Logic Friday (XOR)

- XOR Example



Icarus Verilog

- **Icarus Verilog** is a **Verilog simulation** and **synthesis** tool.
- It supports the 1995, 2001 and 2005 versions of the standard
- It operates as a compiler

Icarus Verilog

- The compiler can generate an intermediate form called **vvp assembly**.
- **vvp** executes the compiled "**vvp assembly**"
  - Writes VCD-format log file as output
- **gtkwave** is an open-source waveform viewer that displays VCD (and other) files graphically
  - http://gtkwave.sourceforge.net/

Verilog Debugging

- Constant syntax:
  - 1'b0 = 1 bit long, binary 0
  - 8'b01011010 = 8'h5A = 8'd90
- $display – like printf in C
  - $display($time, " Count changed to %d", count);
- $dumpfile and $dumpvars
  - dump all signal values to a file for later viewing
    - $dumpfile("output.log")
    - $dumpvars(0, top_level_module)

References

- Linda Null, Julia Lobur, The Essentials of Computer Organization and Architecture
- Mano's textbook Slides